Radio Shack®

Nine Dollars and Ninety-Five Cents

Cat. No. 62-2072

Advanced TRS-80™ Level II BASIC

By
Don Inman
Bob Albrecht
Ramon Zamora

a learning guide

MORE TRS-80 BASIC™

DON INMAN
RAMON ZAMORA
and
BOB ALBRECHT

Dymax Corporation
Menlo Park, California

John Wiley & Sons, Inc.
New York • Chichester • Brisbane • Toronto

Library of Congress Cataloging in Publication Data:

Inman, Don

More TRS-80 BASIC.

(Wiley self-teaching guides)
Includes index.
I. TRS-80 (Computer)—Programming. 2. Basic. (Computer program language) I. Zamora, Ramon. II. Albrecht, Bob, 1930- . III. Title. IV. Series.
QA76.8T18I56 001.64'2 81-150

ISBN 0-471-08010-1 AACR2

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents

TO THE READER

If you bought this book we assume that you already have a TRS-80 microcomputer, or the use of one in a school or office. And we hope that you have read, enjoyed, and learned from our earlier book, TRS-80 BASIC.* (Or another book that introduced you to BASIC and the TRS-80 computer.) This book starts where TRS-80 BASIC ended and will give you more hours of pleasure from your computer than ever before.

We begin with a review of commands and statements and move quickly to a detailed guide to your TRS-80's memory so you can get the most out of PEEK and POKE and use your computer most efficiently. We then progress through Graphics, Cassette and Disk Files, and Animation. Lots of programs demonstrate what is taught and there is plenty of opportunity to change programs, add to them, and write your own. You get some tricks for saving space and many special programs that range from car races to phone indices. And Self-Tests let you check your progress and understanding throughout the book.

If you have read any of our other books you know that we feel computer terminology and concepts can be introduced within a framework of fun and exploration, and that when we do this, *true* learning takes place. Microcomputers are here to stay. Everything in this book is directed toward helping you fully explore the workings and uses of your TRS-80 microcomputer.

The next section tells you How to Use This Book. Why don't you first browse through that, then begin your further adventures with your TRS-80 and BASIC.

*TRS-80 BASIC, Albrecht, Inman, and Zamora, (C) John Wiley and Sons, Inc., 1980.

HOW TO USE THIS BOOK

This book is a Self-Teaching Guide. This means that *you* can use the book to teach *yourself*. Each chapter of the book is composed of short bites of information presenting a single idea or topic on the BASIC language, the TRS-80, a special feature, or a program that is being developed. Throughout the sections there are lots of questions and exercises to be done.

We encourage you to use this book while you are in front of a TRS-80 computer. Try the programs and exercises that are discussed on the machine. Let the TRS-80 be your "teacher."

The first page of each chapter briefly lists what the chapter is to cover. Scan that list. If you feel you already know the material to be covered skip to the back of the chapter and take the Self-Test. You can review the chapter if you have trouble answering the questions.

The material in the book gets more challenging as you move through the chapters in order. If you have only a basic knowledge of BASIC and your TRS-80 start with the first chapters and don't try to skip around. If you know quite a lot of BASIC and are pretty familiar with your computer look at the Table of Contents and feel free to explore some of the later chapters.

As a final note before you start, be aware that as you use the book, enter lines into your computer, answer Self-Test questions, and do the exercises:

YOU CANNOT DO ANYTHING WRONG!

There is no way you can "hurt" or "harm" the computer by what you type into it. You may make *mistakes,* but that is a natural part of learning and exploration. In fact, we introduce deliberate errors in several places as part of the learning process.

So, explore, enjoy, and tell us about your discoveries as you use your Radio Shack TRS-80 microcomputer.

CHAPTER ONE

Introduction

Welcome to this second book in a series designed to help you discover and use more of the features of your Radio Shack TRS-80 Model I computer. In the first book, *TRS-80 BASIC*,* we assumed you were a newcomer to computers and computing. We used a frame-by-frame presentation of material that kept everything down to small, bite-sized lumps. Here, however, we assume that you have finished the first book (or its equivalent) and that you are familiar with some aspects of the Level II BASIC language. The material is presented as a series of chapters that expand on what was covered in the beginner's text. It includes detailed discussions of several new language features of your TRS-80 and self-test exercises and answers at the end of nearly every chapter.

* Albrecht, Inman, and Zamora, *TRS-80 BASIC: A Self-Teaching Guide*. John Wiley & Sons, Inc., N.Y., N.Y., 1980.

In this chapter, we will preview some of the things you will learn and briefly review the commands, statements, terms, and other features of BASIC that we covered in *TRS-80 BASIC*. In chapter 2, we will assume that you are familiar with the elements of BASIC and begin introducing new features of TRS-80 BASIC.

Throughout this book, each new BASIC statement is described in detail when it is first used. Longer programs are broken into logical blocks, or sections, and each section is explained thoroughly. When the user must interact with a program, there are sample program results with sketches of video displays. The material covered is summarized at the end of each chapter.

We have worked to make this book easy to read, to understand, and to use. Some programs appear more than once. These repetitions provide continuity and emphasize relationships between new and previously used instructions. Elementary forms of some programs appeared in our beginners' book.

The *Level II BASIC Reference Manual,* supplied with your TRS-80 Model I computer, is used often as a source for discussions of BASIC statements, commands, and functions. Some Radio Shack hardware and software products are also used as examples.

The central theme of the book is practical, application-based uses of the machine. In some cases, the information is developed in terms of educational or recreational programs. Where possible we indicate several areas where a technique or approach may apply. You will also be introduced to file handling techniques and applications of your TRS-80 cassette recorder and disk. The TRS-80 has so many features and capabilities that we were hard pressed to keep this second book down to its current size.

We do, however, cover new statements, such as PEEK and POKE, the many mathematical and ERROR functions and routines, and most other features of TRS-80 Level II BASIC that were not described in our first book. We include two full chapters of useful, powerful graphic techniques and a chapter on the generation of sound and music, using inexpensive devices that plug into your TRS-80.

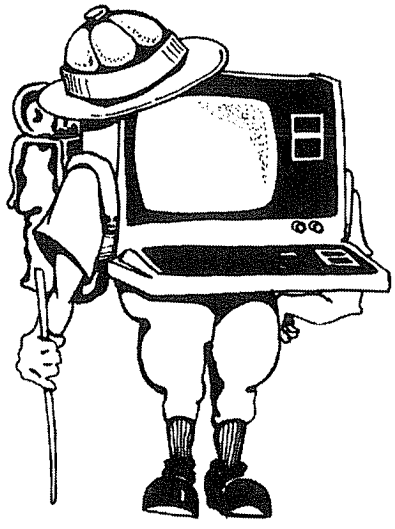
The addition of easy-to-use graphics and sound has lifted computer use out of the number and word crunching business and industrial world and greatly enhanced the computer's recreational and educational aspects. Today's microcomputers, such as the TRS-80, have created an opportunity for nearly everyone to own, use, and master the "mysteries" of a small computer. The small computer is finding its way into the home, the school, and the small business. Thousands of programs are being developed for these machines to perform a wide variety of educational, recreational, and business-related tasks.

With the steady increase of programs and computer users, the number of new applications for these powerful tools will continue to grow. New users bring new interests and avocations that lead to different problems to be solved. Everyone benefits as new solutions are discovered and shared, opening up new areas to apply the tool. The cycle feeds on itself, and everyone has an opportunity to be an inventor and creator in this rapidly growing field.

This book will help you expand your knowledge of Level II BASIC and expand your ability to create programs. As you make greater use of your TRS-80, you will find that you and those around you have both the ability and skill to design programs

that perform any number of tasks you may wish to do. By the time you complete your tour of this book, you will have used nearly *every* BASIC language command and statement available on a Level II 16K machine. For those of you with larger machines (24K, 32K, and so forth), the material in this book will work as well.

As always, we encourage you to venture forward into the book with a spirit of exploration and discovery. We believe that learning to use a small computer can be both fun and educational. In that vein, we occasionally try to lighten things here and there with a bit of humor, and even go so far as to POKE (ooops, there we go — poke) fun at ourselves. Enjoy learning some new ways to use your TRS-80, and share what you learn with those around you. Are you ready? Let's go! If you need to review, move on to the next section. If not, skip the review, and look at chapter 2 for a "guided tour of memory."



Things We Expect You to Know

Although we assume that you have a background knowledge of Level II BASIC, a brief review of the BASIC statements, commands, and functions from the beginner's book is presented here. If you feel confident in your knowledge of Level II BASIC, you can either pass quickly through the review section or skip on to chapter 2.

BASIC Statements and Commands

ASC(*string*) Returns the ASCII code of the first character in the string argument specified in parentheses.

CHR$ (*exp*) Returns a one-character string of *exp*, which may be an ASCII, control, or graphics code.

Clear n Sets numeric values to zero and strings to null; sets aside n bytes of string space in memory.

CLS Clears the video display.

DATA item list Holds the data for access by a READ statement; items may be numeric or string; items are separated by commas.

DELETE *mm-nn* Deletes the program lines from line *mm* through *nn*.

DIM array (*dim#1, dim#2, . . .dim #k*) Reserves storage for a k-dimensional array with the specified size per dimension; DIM may be followed by a list of arrays separated by commas.

EDIT *mm* Puts you in the edit mode at line *mm*.

EDIT Puts you in the edit mode at the last line entered, altered, or in which an error has occurred.

END Ends the execution of a program.

FOR *var* = *exp* TO *exp* STEP *exp* Opens a FOR-NEXT loop. STEP is optional — if not used, STEP will be 1.

GOSUB *mm* Branches to the subroutine beginning at line *mm*.

GOTO *mm* Branches to the line number specified by *mm*.

IF *exp* THEN *statement #1* ELSE *statement #2* Tests the expression, *exp*; if True, statement #1 is executed and control proceeds to the next program line (unless statement #1 is a GOTO); if *exp* is False, statement #2 is executed; the ELSE statement #2 is optional.

INKEY$ Reads the keyboard and returns a one-character string, which is the string value of the key that was last pressed.

INPUT "*message*"; *var* PRINTs the message (if any) and waits for an input from the keyboard; *var* may be a numeric or string variable; a list of variables separated by commas may be used.

INT (*exp*) Returns the largest integer that is not greater than the expression, *exp*.

LEFT$ (*string, n*) Returns the first n characters of the specified string.

LEN (*string*) Returns the length (number of characters) of the string.

LET *var* = *exp* Assigns a value equal to *exp* to the specified variable, *var*.

LIST Lists the entire current program in memory.

LIST *mm-nn* Lists only lines *mm* through *nn* of a program; *-nn* is optional.

MID$ (*string, p, n*) Returns a substring of the specified string; the length of the substring will be *n* starting with the character in position *p*.

NEW Deletes the current program in memory and resets all variables, pointers, etc.

NEXT *var* Closes a FOR-NEXT loop; the variable, *var*, is optional.

POS (0) Returns a number indicating the current cursor position; the argument (0) is a dummy variable.

PRINT *exp* Output the value of *exp* to the display; *exp* may be numeric or string variable or constant, or a list of such items.

PRINT @*n* Modifies the PRINT statement to begin printing at the display position *n*.

RANDOM Reseeds the random number generator (starts with new number).

READ *variable list* Assigns values to the specified variables starting with the current data item from the DATA statement.

REMARK Remark indicator; tells the computer to ignore the rest of the current line; abbreviation REM may be used also.

RESET (*X, Y*) Turns off the graphic blocks at horizontal coordinate *X* and vertical coordinate *Y*.
RESTORE Resets the data pointer to the first item in the first DATA statement.
RETURN Branches to the statement following the last executed GOSUB.
RIGHT$ (*string, n*) Returns the last *n* characters of the string specified.
RND (0) Returns a pseudorandom number between 0.000001 and 0.999999, inclusive.
RND (*exp*) Returns a pseudorandom integer between 1 and INT (*exp*), inclusive.
RUN *mm* Executes a program beginning at line *mm*; *mm* is optional; if not specified, the execution begins at the lowest numbered line.
SET (*X, Y*) Turns on the graphic block at horizontal coordinate *X* and vertical coordinate *Y*.
STRING$ (*n, char*) Returns a sequence of *n* character symbols using the first character of *char*.
TAB *n* Modifies a PRINT statement; moves cursor to the specified display position (0 through 63) on a given line.
TROFF Turns off the trace function.
TRON Turns on the trace function that displays the line number of each line executed.
VAL (*string*) Returns the number represented by the characters in the specified string.

Frequently Used Terms

Backspace key Used to move the cursor left one space and erase the character in that space. If the SHIFT key is held down at the same time, an entire line is erased.
Break key Stops the computer at the current line being executed.
Cassette recorder Used to store data outside the computer.
Clear key Clears the TV display.
Concatenation Joins two or more strings together.
Cursor Shows where the next printed character will appear.
Data pointer A pointer (or counter) the computer uses to tell it which item to select from a data list.
Debug To remove errors, or "bugs," from programs.
Edit mode Allows changes in an existing program without retyping the whole line.
Empty string("") A string with nothing in it, not even a space.
Enter key Tells the computer a command or statement is completed.
Flag A signal to the computer that some process has been completed and should be discontinued.
Floating point A special method of writing very large or very small numbers.
Inequality symbols Symbols used to express a relationship between two quantities that are not necessarily equal.
Keyboard The control center of the TRS-80.
Line number Used to tell the computer which line to execute next.
Multiple-statement line A program line that contains more than one statement to be executed.

Numeric variable A variable that identifies a number.
One-dimensional array A list of numbers or strings arranged in a specific order.
Order of operations The order in which arithmetic operations are performed within a given arithmetic expression.
Power of numbers Tells how many times a number is to be used as a factor.
Power supply The TRS-80 power supply converts 110- to 120-volt AC to smaller DC voltages.
Print positions Positions on the video display that are numbered from the upper left to the lower right (0 through 1023).
Prompt The > symbol tells you that it's your turn to tell the computer what to do.
RAM (Random Access Memory) Used to store your programs and data.
Random number A number chosen from a given set of numbers so that each number has an equal chance of being selected.
Ready A message printed on the video screen that indicates the computer is ready to do something.
ROM (Read Only Memory) Used to store the TRS-80 operating system and the BASIC interpreter.
Rounding numbers Changing values to the nearest specified decimal place.
Sequence of values Values that come one after another in order.
String A string of numerals, letters, and/or special characters.
String variable A variable that identifies a particular string.
Superscript Specifies the power of a number.
Two-dimensional array A table of numbers or strings consisting of rows and columns.
Variable subscript Denotes the order of an element in an array named by the variable.
Video display The TV screen on which graphics and/or text is shown.

CHAPTER TWO

A Guided Tour of Memory

The TRS-80 performs its wondrous feats with a carefully planned, ingenious use of memory. Almost everything that goes on inside the computer uses memory in some way. The computer's operating procedures, the BASIC programs you write, and the data used by your programs are all stored in memory.

Did you ever wonder why you lose your programs when the computer is turned off, yet the computer is still able to operate when it is turned on again? If the computer "forgets" your programs when turned off, how can it "remember" its own operating procedures? Stay with us. This chapter presents a map to guide you through TRS-80 Memory Land. You will see the different kinds of memory and learn how each is used. When you finish the chapter, you'll be able to PEEK and to POKE around inside your own memory. You'll also have a better understanding of how the computer works and how you can use it more efficiently.

You will learn:

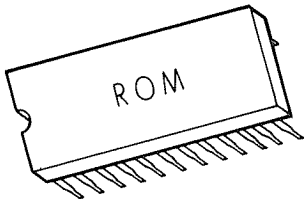
- how ROMs and RAMs are packaged,
- how the CPU works with ROM,
- how to PEEK into ROM,
- the difference between ROM and RAM,
- how ROM and RAM are organized,
- how your BASIC program uses RAM,
- why the MEMORY SIZE? prompt is used,
- how memory space is reserved for strings,
- how to conserve memory space,
- how to PEEK and/or POKE into RAM,
- how to use POKE with care, and
- where the video screen is located and how to use it.

What is Memory?

The first chapter of *TRS-80 BASIC** briefly discusses two kinds of memory—ROM and RAM. The computer can read information from ROM, but cannot erase or change it in any way. Therefore, this type of memory is called *Read Only Memory,* or ROM for short. Information in ROM is *permanently* stored, much like the information on the pages of this book or the information on a phonograph record. It cannot be changed and it is not "forgotten" when the computer is turned off.

A ROM in a computer can be replaced, just as a book is replaced by a new, revised edition or a record is replaced in your phonograph. But the information in that particular ROM cannot be changed from the keyboard or by a BASIC program.

The ROMs inside the TRS-80 are enclosed in rectangular packages with twenty-four small, metal legs, twelve on each side. These legs make electrical connections to the rest of the computer.



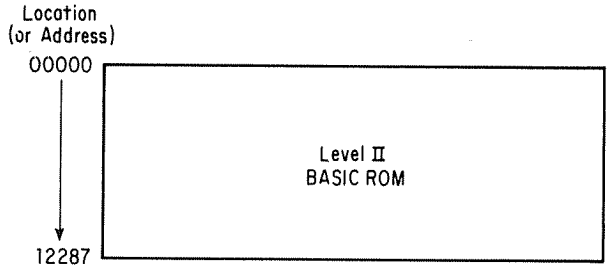
The heart of the computer is called the *Central Processing Unit* (CPU). It is contained in a package similar to the ROM's. Although the CPU does most of the work (or processing) for the computer, the ROMs are the boss; they tell the CPU what to do, how and when to do it, and where to put the results.

When the TRS-80 is turned on, the CPU immediately looks at a certain memory location in ROM to see what it should do. The ROM then takes over, giving directions to "fire up" the system and get it READY for your input.

Since the ROM is so important to the computer, it is placed at the beginning of the memory block. Regardless of whether you have a 4K, 16K, 32K, or other size TRS-80, *the ROM occupies the memory locations numbered from 0 through 12287.* Therefore, every TRS-80 Level II computer has a 12K ROM (K for thousand). Each K actually refers to 1024 locations.

*Albrecht, Inman, and Zamora, *TRS-80 BASIC: A Self-Teaching Guide.* John Wiley & Sons, Inc., N.Y., N.Y. 1980.

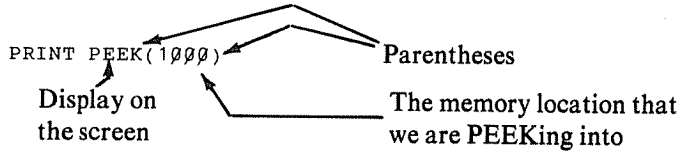
$12K = 12 \times 1024 = 12288$ (Numbered 0 through 12287)



Remember, you can only read information *from* ROM. Nothing can be written into the memory locations from 0 through 12287.

PEEK into ROM

You can read information from ROM using Level II BASIC's PEEK function. This function lets you PEEK at (or read) the information stored in one memory location. PEEK can be used with a PRINT statement in the Immediate Mode, as well as within a program.



Now, turn on your TRS-80 and PEEK around a bit.

- When it says MEMORY SIZE? you press ENTER

This is what you see:

```
MEMORY SIZE?
RADIO SHACK LEVEL II BASIC
READY
>-
```

- You type: PRINT PEEK(1000) and press ENTER

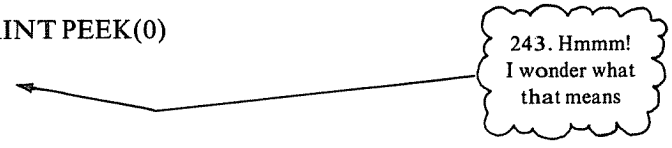
This is what you see:

```
MEMORY SIZE?
RADIO SHACK LEVEL II BASIC
READY
  PRINT PEEK(1ØØØ)
  56
READY
>-
```

By PEEKing, we found that the number 56 is stored in ROM location 1000. Hmmm, wonder what is in location 0.

- You type: PRINT PEEK(0)

  It prints: 243



So far, you've found a number (243) in memory location 0 and another number (56) in memory location 1000. Is that all you're going to find — a bunch of numbers? Try another. This time find out if the PEEK statement can use a variable instead of the numeric location.

- You type: A = 1
- You type: PRINT PEEK(A)

  It prints: 175 (This number is in memory location 1, since A = 1.)

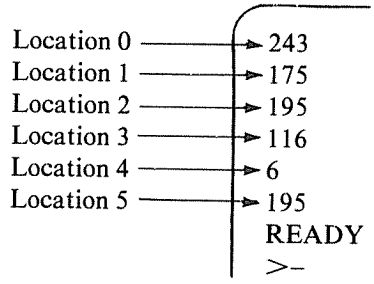
You now know this about the Level II ROM:

Location	Value
0	243
1	175
•	•
•	•
•	•
1000	56

To find what is in the first few locations, type in the following single line and press ENTER:

```
CLS: FOR A=Ø TO 5: PRINT PEEK(A): NEXT A
```

Lo and behold! You get the numbers stored in memory locations 0 through 5.



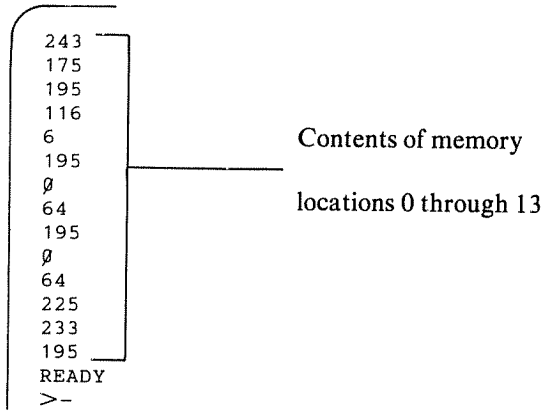
That one line in the Immediate Mode performs the same function as the following BASIC program:

```
1Ø CLS
2Ø FOR A = Ø TO 5
3Ø PRINT PEEK(A)
4Ø NEXT A
```

Try using the program and see if you get the same results. To look at more results, change line 20. If you try to get too much data on the screen, the results will scroll by so fast that you can't see them all. Try changing line 20 to:

```
2Ø FOR A = Ø TO 13
```

Now run the program again and compare your results with ours.



By this time, you may be convinced that all ROM contains is a bunch of numbers. What do the numbers mean? They don't mean anything to you, but they *do*

mean something to the computer. It interprets them as specific instructions—what, how, and when to do something.

Perhaps you are itching to PEEK into the rest of the ROM. The following program lets you pick the starting and ending locations for a screenful of ROM locations. The addresses of the locations where the values are stored will also be printed.

Use the following program to explore ROM:

ROM PEEK Program

```
100 REM * INPUT LOCATIONS *
110 CLS
120 INPUT ''START ADDRESS'';B
130 INPUT ''END ADDRESS'';C
140 CLS

200 REM * PRINT CONTENTS OF ROM *
210 FOR A = B TO C
220 PRINT A, PEEK(A)
230 NEXT A

300 REM * GO BACK FOR MORE *
310 GOTO 120
```

Be sure to look at the last locations in the ROM when you have entered the ROM PEEK Program. Are all of the ROM locations used? Decide for yourself by running the program with the following inputs:

```
START ADDRESS? 12275
END ADDRESS? 12288
```

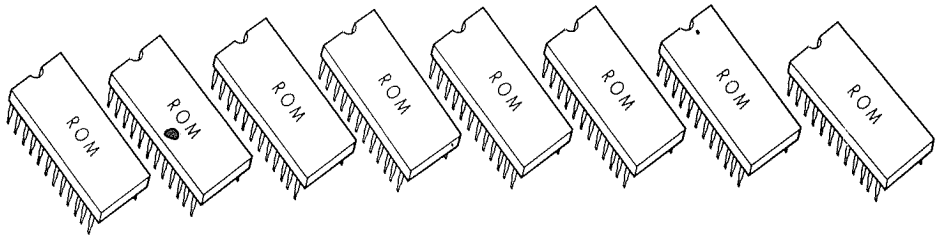
After entering these starting and ending addresses and pressing ENTER, this is what you will see:

```
12275          195
12276          152
12277          26
12278          193
12279          209
12280          195
12281          25
12282          26
12283          0
12284          0
12285          0
12286          0
12287          0
12288          255
START ADDRESS? -
```

The last five locations in ROM (12283 through 12287) contain zeroes. Location 12288 is actually the beginning of RAM, which we will discuss next.

Random Access Memory (RAM)

The ROMs and the CPU are physically located inside your TRS-80 keyboard. The keyboard is also the home of many electronic circuits and integrated circuit "chips." Eight of these chips, enclosed in packages similar to the ROM chips, make up the first 4K or 16K of your *Random Access Memory* (or RAM).



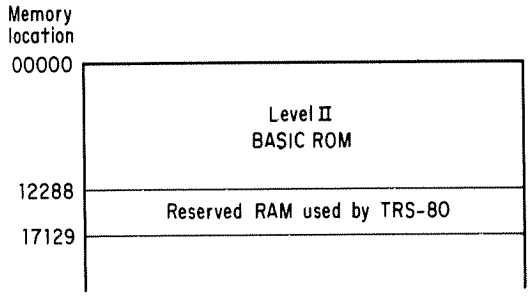
Each RAM chip holds either 4,096 bits (*binary digits*) or 16,384 bits, depending on the computer's memory size. In a 4K machine, each chip holds 4,096 bits; in a 16K machine, each chip holds 16,384 bits. The computer combines 1 bit of information from each chip to make a "byte" of information. The TRS-80 is called an 8-bit computer because it works with pieces of information 8 bits long. A binary number whose length is 8 bits is called a byte. One memory location can hold 1 byte of information made up of 1 bit from each of the 8 chips.

RAM is different from ROM in two important ways.

1. *Information can be written into, stored into, or read from RAM.* Therefore, you can erase or change the information in RAM. It is like a blackboard or scratch pad.
2. *All information is lost from RAM when the electric power is removed from the chips.* The information stored there is *not* permanent, unlike ROM. That is why the computer "forgets" your BASIC programs when it is turned off. BASIC programs are stored in RAM.

The TRS-80 uses RAM for many of its own operations. The area of RAM reserved for this use is located at address numbers 12288 (just above the ROM) to 17129. This area is *reserved* for special purposes, such as keyboard memory, video memory, and other duties called "household chores." It is not used to store BASIC programs, but is used by the computer as it stores or runs your BASIC programs.

You can see in the diagram below that the computer's memory is growing.



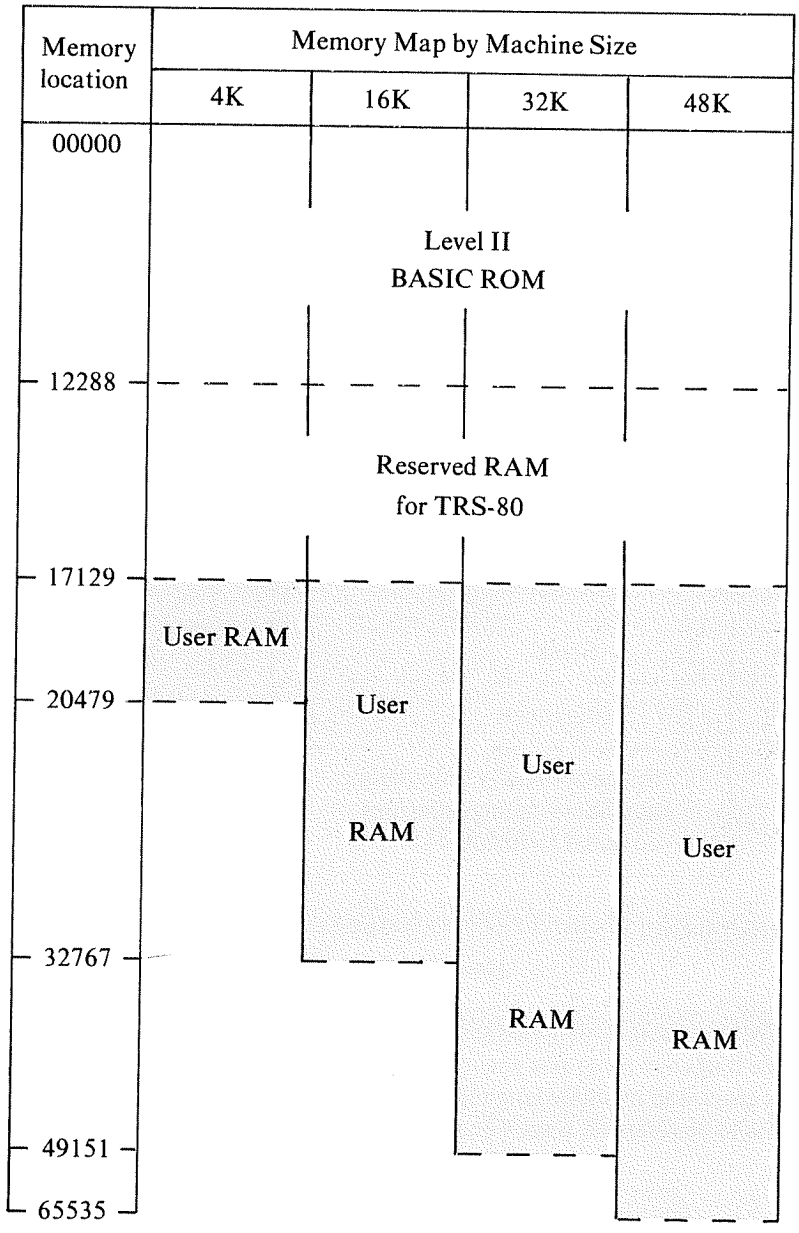
You've finally reached the area of RAM *that you can use for your programs.* This area begins at memory location 17129. The highest memory location available to you depends upon the size of your computer. The upper level address is 20479 for 4K, 32767 for 16K, 49151 for 32K, and 65535 for 48K computers. The table below shows the RAM areas for the different size systems and the amount of RAM that may be used by your programs.

	Computer Size			
	4K	16K	32K	48K
Top location	20479	32767	49151	65535
Low location	17129	17129	17129	17129
Usable memory	3350	15646	32022	48406

Memory use by computer size

Notice that the amount of usable memory may not be what you would expect. The TRS-80 uses some of your RAM for various necessary purposes.

A more dramatic display of the amount of RAM available to you, the user, can be shown graphically. The following diagram gives a picture of the RAM capacities for different computer systems.

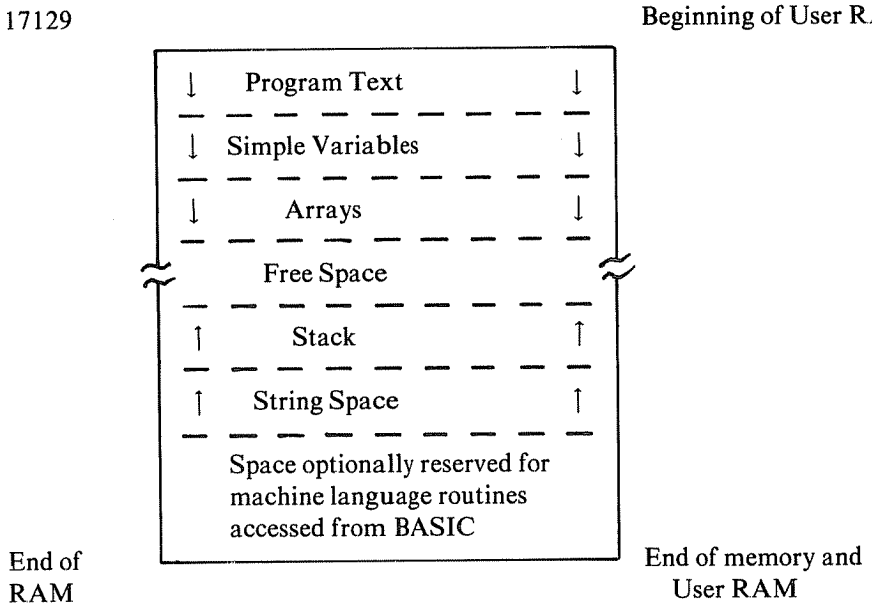


TRS-80 level II memory map

How RAM Is Used

RAM is used in many interesting ways. As you type in the text of your program, RAM is used from location 17129 *upward*. String space is assigned from the top of your RAM *downward*. Remember, string space is automatically set for 50 bytes when you turn the system on. The amount of string space can be changed by using the CLEAR N statement (where N is the number of locations to be reserved for string characters). Simple variables are stored upward, immediately following the program text. Any arrays used are assigned upward following the simple variables.

Next in line are locations used by *the stack*. The stack keeps track of data during GOSUB statements and FOR-NEXT loops. Therefore, your RAM is gobbled up from both ends. The TRS-80 munches its way toward the middle of RAM, as indicated in the following table.



How RAM is used

Notice that there is an optional space at the top of memory that can be reserved for machine language programs you may want to access from your BASIC program. Use of this space and machine language programs is described in chapter 8. This space, if it is to be reserved, is set when you first turn the computer on. It responds:

```
MEMORY SIZE?-
```

If you are not going to use a machine language program (and usually you won't), the space is *not* reserved when you press the ENTER key.

```
MEMORY SIZE? <-------- Press ENTER
RADIO SHACK LEVEL II BASIC
READY <-------- Then READY message is printed
>-
```

If you are going to access a machine language program from a BASIC program, type in the address of the start of the machine language program following the MEMORY SIZE? prompt before pressing the enter key. The computer then reserves (or saves) the locations from the stated memory location to the top of your RAM for the machine language program.

Example: A machine language program is to be accessed from BASIC starting at location 32700.

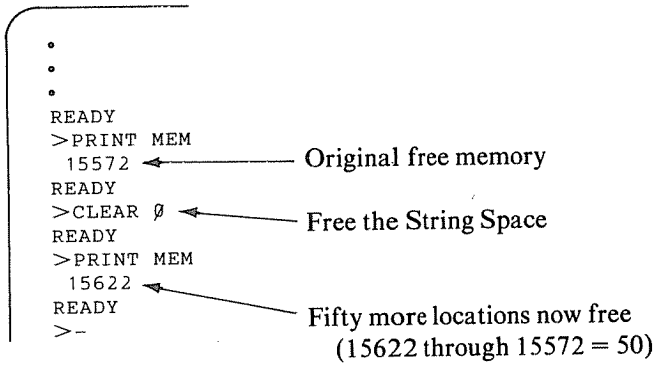
```
MEMORY SIZE?32700
RADIO SHACK LEVEL II BASIC
READY
>-
```

Assume that you are not using a machine language program and you have just received the READY prompt after turning on your 16K machine.

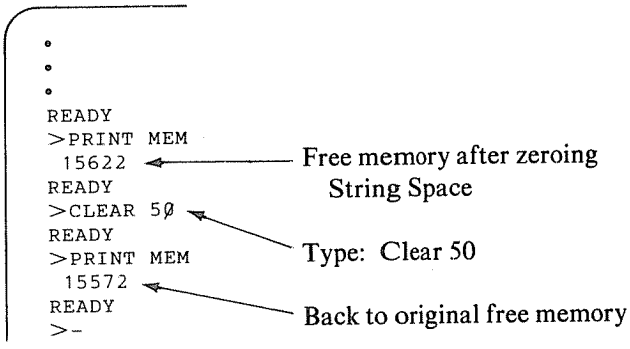
- Next, verify the amount of "free" memory space in your computer.
- Type: PRINT MEM

```
MEMORY SIZE?
RADIO SHACK LEVEL II BASIC
READY
>PRINT MEM <-------- You type
 15572 <-------- Original free memory
READY
>-
```

Suppose you know that you will not be using any strings. Can you *unreserve* the string space that is saved? To find out, use the CLEAR N statement with N being 0. Then, PRINT MEM again.



Now set the string space back where it was originally.



Next, put in a small program one line at a time and match the free memory space decrease as the program text is entered.

Assume, as you enter this program, that you are using a 16K TRS-80. You have just turned the computer on and it shows free memory space of 15572 locations.

```
READY
>10 A=5
>PRINT MEM
 15564
READY
>-
```

First program line

Eight locations used up for line 10 (15572 through 15564)

```
READY
>10 A=5
>PRINT MEM
 15564
READY
>20 B=6
>PRINT MEM
 15556
READY
>-
```

Second program line

Eight more locations used for line 20 (15564 through 15556)

```
READY
>10 A=5
>PRINT MEM
 15564
READY
>20 B=6
>PRINT MEM
 15556
READY
>30 PRINT A+B
>PRINT MEM
 15546
READY
>-
```

Third program line

Ten more locations used for line 30 (15556 through 15546)

You used 26 memory locations to store this three-line program. When you RUN the program, more memory space is used to store the variables. RUN the program and see how much total memory is used.

```
.
.
.
>PRINT MEM
 15546
READY
>RUN
 11
READY
>PRINT MEM
 15532
READY
>-
```

5+6 = 11

Fourteen more locations used for the run (15546–15532)

14+26 = 40 total locations

You can see that memory gets used up pretty fast as you type in each line of the program. Remember, you use memory to store program text, variables, arrays, strings, etc. Even line numbers and spaces between words in the text take up memory locations. To demonstrate, type the following three-statement program on one line with no spaces. Then check the free memory again.

- Type: NEW
- Then enter the program on one line.

```
READY
>10 A=5:B=6:PRINT A+B
>RUN
 11
READY
>PRINT MEM
 15541
READY
>-
```

← The program: Three statements on one line

← 15572−15541 = 31 locations for the program and run

This one-line program used 31 locations. Therefore, you saved a total of 9 locations by squeezing data onto one line. You *can* save memory in this way, but it makes a program hard to read.

PEEK into RAM

Would you like to see what a program looks like after it's stored in RAM? Since you know the program is stored in RAM starting at memory location 17129, you can PEEK into that area of memory after you have entered your program. Use the previous three-line program because you know how much memory it used. If it's not still in your computer, type NEW and enter it again.

```
READY
>10 A=5
>20 B=6
>30 PRINT A+B
>PRINT MEM
 15546
READY
>-
```

Since the program took 26 locations (15572–15546), you want to look at memory locations 17129 through 17154. However, if you add a couple more locations you will see something significant.

```
READY
>10 A=5
>20 B=6
>30 PRINT A+B
>PRINT MEM
 15546
READY
>FOR C=17129 TO 17156:PRINT PEEK(C);:NEXT C
 241 66 10 0 65 213 53 0 249 66 200 66 213 54 0
 3 67 30 0 178 32 65 205 66 0 0 0
READY
>-
```

End of program

That long list of numbers is the text of the three-statement program. The three zeros at the end signify the end of the program text. Amazingly, the computer can interpret those numbers meaningfully. How does it do it? Well, it's that work horse, CPU (Central Processing Unit), following the directions of its boss, the ROM. Together they make a great pair, figuring out those crazy codes and performing the work with lightning speed.

If you added another instruction to your program, you'd see that the program in memory has been lengthened. Try adding an END statement at line 40.

```
.
.
.
>40 END
>PRINT MEM
 15540
READY
>-
```

Six more memory locations have been used for line 40 (15546 through 15540). Therefore, look at locations 17129 through 17162 this time.

```
READY
>FOR C=17129 TO 17162:PRINT PEEK(C);:NEXT C
 241 66 10 0 65 213 53 0 249 66 20 0 66 213 54 0
 3 67 30 0 178 32 65 205 66 0 9 67 40 0 128 0 0
 0
READY
>-
```

If you compare the two programs, you can see that 6 more locations containing the codes 0, 9, 67, 40, 0, and 128 have been added just before the three zeros that indicate the end of the program.

One last demonstration before leaving the subject of program RAM. This time enter a program that will PEEK at *itself* when it is run.

Once again, the program will start at location 17129; ROM sees to that. Type NEW and enter the following program:

```
READY
>10 FOR A=17129 TO 17172
>20 PRINT PEEK(A);
>30 NEXT A
>RUN
 255 66 10 0 129 32 65 213 49 55 49 50 57 32 189
 32 49 55 49 55 50 0 11 67 20 0 178 32 229 40 65
 41 59 0 19 67 30 0 135 32 65 0 0 0
READY
>-
```

End of program

To get the same results, the program must be entered exactly as shown. Any extra spaces change the codes produced. Try putting spaces on both sides of the equal sign in line 10. Then run the program again. Compare the two results. See the difference? Try other changes and compare results. Be careful, though. Don't let your curiosity get the best of you — the habit is catching. You can spend hours at this sort of thing. You might even discover what some of those numbers mean to the computer.

PEEK and POKE

The PEEK instruction is very useful. It offers a passive way to investigate what is inside the computer's memory without disturbing what is there. PEEK has a companion, named POKE, that is just the opposite. The POKE instruction is very active. *It changes what is inside a specific memory location.* Therefore, use it with great care. Its use can alter a vital piece of information if used at the wrong time or the wrong place. It works like this:

POKE 17500,10

POKE a new value. Into location 17500. This is the value POKEd.

POKE and PEEK are often used together. It is often desirable to PEEK at the value in a memory location *before* you POKE in a new value. Jot down on paper what is there. Then, if a disaster occurs when the new value is POKEd in, you will know what value to POKE back in to restore the original condition.

Get your computer READY to try a few PEEKs and POKEs.

Try a POKE into ROM

Remember, the TRS-80 ROM, discussed in the early part of this chapter, is addressed from 0 through 12287.

- First, PEEK into the ROM to see what is in location 120.

```
READY
>PRINT PEEK(12Ø)
 33
READY
>-
```

- Then, try to POKE the number 5 into location 120.

```
READY
>PRINT PEEK(12Ø)
 33
READY
>POKE 12Ø,5 <------ POKE it!
READY
>-
```

- Now, PEEK again to see if it changed.

```
READY
>PEEK(12Ø)
 33 <-------------------
READY                   |
>POKE 12Ø,5             |
READY                   |
>PEEK(12Ø)              |
 33 <-------------------  What happened?
READY                      It didn't change.
>-
```

Remember, ROM is *Read Only Memory*. It cannot be written into. The POKE statement writes data *into* memory. You can't do that with ROM. PEEK reads, and POKE writes. The TRS-80 accepts the POKE 120,5 statement and tries to write 5 into location 120. Since location 120 is a part of ROM, it cannot be written into. No error prompt will be displayed but the data has not been accepted.

If you or your TRS-80 seem hopelessly confused, turn off the TRS-80, then turn it on and start over!

POKE into RAM

Although you can't POKE into ROM, you *can* POKE into RAM. *Random Access Memory* can be read from or written into. For the time being, let's stay in the user RAM area from location 17129 up. Erase any program that may be in your TRS-80 by typing NEW. Then PEEK at 17150, POKE a 10 into 17150, and then PEEK at 17150 again.

```
READY
>PRINT PEEK(17150)
 0
READY
>POKE 17150,10
READY
>PRINT PEEK(17150)
10  <-------- It's there!
READY
>-
```

Success! You can POKE into RAM. Now, try this:

- POKE the numbers 10, 11, and 12 into locations 17150, 17151, and 17152.

```
READY
>POKE 17150,10
READY
>POKE 17151,11
READY
>POKE 17152,12
READY
>FOR X=17150 TO 17152: PRINT PEEK(X): NEXT X
10
11  <-------- There they are
12
READY
>-
```

Recall the three-line program on page 20:

```
10 A=5
20 B=6
30 PRINT A+B
```

Enter the program in the computer again, and then PEEK at locations 17129 through 17156.

```
READY
>10 A=5
>20 B=6
>30 PRINT A+B
>FOR C=17129 TO 17156:PRINT PEEK(C);:NEXT C
 241 66 10 0 65 213 53 0 249 66 20 0 66 213 54 0
 3 67 30 0 178 32 65 205 66 0 0 0
READY
>-
```

Enter program

Peek

Keep your eye on this one; it is the number in location 17135.

Now, you are going to do something that *you should never*, never do. You're going to POKE a number into the memory area where your program is stored.

- Add line 25, which POKEs the value 54 into memory location 17135 — the memory location circled in the previous screen picture.

```
READY
>25 POKE 17135,54
>LIST
10 A=5
20 B=6
25 POKE 17135,54
30 PRINT A+B
READY
>-
```

There it is

- Now RUN the program.

```
.
.
.
>LIST
10 A=5
20 B=6
25 POKE 17135,54
30 PRINT A+B
READY
>RUN
 11
READY
>-
```

Looks just fine

It looks as if nothing is wrong. Everyone knows that 5 plus 6 is 11. Just to make sure, list the program again.

```
>LIST
10 A=5
20 B=6
25 POKE 17135,54
30 PRINT A+B
READY
>RUN
 11
READY
>LIST
10 A=6
20 B=6
25 POKE 17135,54
30 PRINT A+B
READY
>-
```

—We started with this program

— It worked OK

—We ended up with this program. Now A=6. WHY? Read on.

By using the POKE statement in line 25, you actually changed the program. At line 10, A was set to 5. At line 20, B was set to 6. When the program was executed, these values were stored in the memory area assigned to variables. Line 25 actually changed line 10 in the program. However, the program has already executed that line, so no harm was done to the result produced at line 30. Execute the program again to see what the answer is.

```
.
.
.
READY
>RUN
 12
READY
>-
```

— Sure enough, the answer is now 12

This example produces a very minor fault. You were lucky. You might have destroyed your whole program. REMOVE LINE 25 IMMEDIATELY. Then change line 10 back to A=5.

```
READY
>25
>10 A=5
>RUN
 11
READY
>-
```

— Delete line 25

— Change line 10

— OK!

To emphasize the care that must be used with POKE, we encourage you to produce a major disaster by poking several numbers into the memory area that your program occupies. We think you will discover why this kind of poking is a NO, NO.

- List the program to make sure you have the original program, then add line 40, below.

```
4Ø POKE 17159,187:POKE 1716Ø,Ø: POKE 17161,Ø POKE 17162,Ø
```

```
.
.
.
>LIST
1Ø A=5
2Ø B=6
3Ø PRINT A+B
READY
>4Ø POKE 17159,187:POKE 1716Ø,Ø:POKE 17161,Ø:POKE  17162,Ø
>-
```

Now RUN the program.

```
>LIST
1Ø A=5
2Ø B=6
3Ø PRINT A+B
READY
>4Ø POKE 17159,187:POKE 1716Ø,Ø:POKE 17161,Ø:POKE  17162,Ø
>RUN
 11
READY
>-
```

Nothing strange seems to have happened. RUN it again to make sure everything is all right.

```
READY
>RUN
>-
```

Nothing!

- LIST the program.

```
READY
>LIST
READY
>-
```

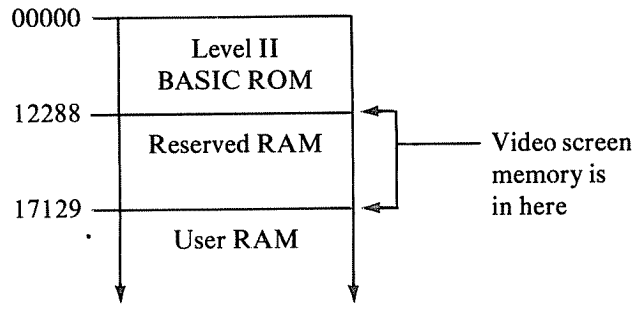
It's gone!

The program destroyed itself. This can happen when the POKE instruction is not used wisely. Results can be altered, a program can be destroyed, or the computer can become so confused that it doesn't know what to do next.

Enough for destructive uses of POKE. Let's now turn to a more useful area of memory and POKE around.

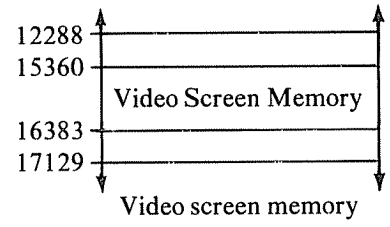
Video Screen Memory

Hidden away in an area called *reserved RAM* is a section the TRS-80 uses for its video display.



Reserved RAM area

The video screen memory consists of 1024 RAM locations, numbered from 15360 to 16383, inclusive.



Video screen memory

This area of RAM memory displays the text and results of BASIC programs. You can alter the screen by poking certain values into the video screen memory (locations 15360 through 16383).

- For example, try this program:

```
READY
>1Ø CLS
>2Ø POKE 15360,191
>6Ø GOTO 6Ø
>RUN
```

You see this:

Press the BREAK key to stop the program.

- Now, add line 30.

```
BREAK, IN 6Ø
READY
>3Ø POKE 15423,191
>RUN
```

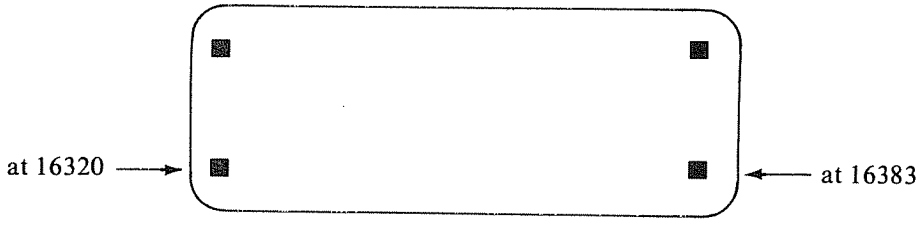
What do you see?

You see this:

- Again, press the BREAK key to stop the program. Then add two more lines.

```
BREAK IN 6Ø
READY
>4Ø POKE 1632Ø,191
>5Ø POKE 16383,191
>RUN
```

Now, you see:



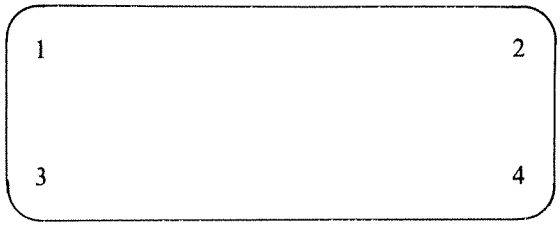
You have now located the memory locations assigned to the four corners of the video screen. Each rectangle occupies the space assigned to one memory location. You can put a number into each corner with the following program. Lines 20,30,40, and 50 POKE the ASCII codes for the numbers 1,2,3, and 4 into the video screen memory.*

*For a review of the ASC function, see Albrecht, Inman, and Zamora, *TRS-80 BASIC: A Self-Teaching Guide*. John Wiley & Son, Inc., N.Y., N.Y., 1980.

- Enter this:

```
READY
>1Ø CLS
>2Ø POKE 15360,ASC("1")
>3Ø POKE 15423,ASC("2")
>4Ø POKE 16320,ASC("3")
>5Ø POKE 16383,ASC("4")
>6Ø GOTO 6Ø
>RUN
```

Then you see:



The next example uses strings, string functions, ASC, and MID$ to poke the numbers 1,2,3,4, and 5 into the first five video screen locations.

```
READY
>1Ø CLS
>2Ø A$="12345"
>3Ø FOR A = 1 TO 5
>4Ø B$ = MID$(A$,A,1)
>5Ø POKE 15359+A,ASC(B$)
>6Ø NEXT A
>7Ø GOTO 7Ø
>-
```

POKE 1,2,3,4, and 5 into locations 15360, 15361, 15362, 15363, and 15364

RUN the program.

```
12345
```

- Sit back and watch the next program POKE zeros into each successive memory location on the top line of the video screen. You can see that the memory locations are numbered from the upper left corner horizontally across the screen.

15360 ——————→ 15423

```
READY
>10 CLS
>20 FOR X = 15360 TO 15423
>30 POKE X,ASC("0")
>40 NEXT X
>50 GOTO 50
>RUN
```

POKE can use a variable address

The RUN:

```
0000000000000000000000000000000000000000000000000000000000000000
```

64 zeros on one line

- To show two lines on the screen, change line 20 to:

```
20 FOR X = 15360 TO 15487.
```

RUN the program again.

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Two rows of 64 zeros

- For a full screen of zeros, change line 20 to:

```
20 FOR X = 15360 TO 16383
```

Now, you'll see the consecutive locations filled with zeros. Add line 50: 50 GOTO 10 and the screen clears after it is full of zeros. Then it all repeats.

Summary

You have explored the TRS-80 memory in this chapter. You learned that:

- ROM (*Read Only Memory*) is located at the low memory locations. It contains the TRS-80 operating system and the BASIC language interpreted. You can PEEK at the contents of ROM, but you cannot POKE new data into it.
- The CPU (*Central Processing Unit*) does most of the work, as directed by the instructions that are located in ROM.
- ROM contains information in the form of numbers that the CPU interprets as specific instructions — what to do and how and when to do it.

- RAM (*Random Access Memory*) is made up of integrated circuits in which data can be stored. RAMs are organized in the TRS-80 to handle bytes of information made up of 8 binary digits.
- You can PEEK at the data in RAM, and also POKE new data into it.
- All information in RAM is lost when the power is turned off, but information in ROM is not lost.
- Some RAM is reserved for special purposes (keyboard, memory, etc.). BASIC programs use RAM memory to store program text, variables, arrays, strings, etc.
- The MEMORY SIZE? prompt may be used to save memory space for machine language programs.
- PEEK and POKE instructions allow you to explore memory and alter RAM memory. POKE must be used with care. It may change data that is important to the operation of your program.
- The POKE statement may be used to create and change the video display, since the display is controlled by the data in a specific block of RAM.

Keeping all these things in mind, try your luck with the following exercises.

Self-Test

1. Complete the sentence: TRS-80 Level II BASIC is stored in the 12288 ROM memory locations, which have addresses _____ through _____ .
2. What does the following statement tell the TRS-80 to do?

   ```
   PRINT PEEK(1234)
   ```

   Answer: _____
3. What does the following statement tell the TRS-80 to do?

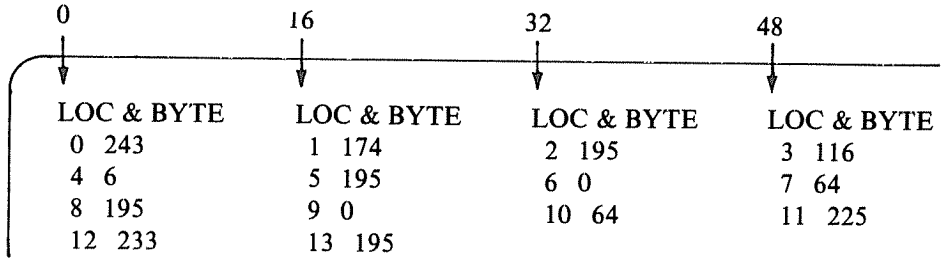
   ```
   X = PEEK(1234)
   ```

   Answer: _____
4. Write a program to explore ROM. When someone RUNs your program, it should begin like this.

   ```
   START ADDRESS?Ø      Enter 0
   END ADDRESS? 13      Enter 13
   ```

After you enter 13 and press ENTER, this should happen:

EACH PAIR OF NUMBERS (LOCATION AND BYTE)
BEGINS AT A STANDARD
SCREEN POSITION — COMMA SPACING, 0, 16, 32, 48.



Your program packs four pieces of information onto each line. Each item consists of the ROM location (LOC) and the number stored there (BYTE).
You decide how to end your program. Simply stop (TRS-80 prints READY and prompt)? Or, use an "idling" GOTO, such as 710 GOTO 710? Or, use INKEY$? Or, ??? *You* choose!

5. Where does TRS-80 reserved RAM begin (what address or location number)?
   _____
6. Where does RAM, *that you can use,* begin (address or location number)?
   _____
7. Where does usable RAM end?
   a) 4K TRS-80 _____
   b) 16K TRS-80 _____
   c) 32K TRS-80 _____
   d) 48K TRS-80 _____
8. What does the following statement tell the TRS-80 to do?

```
POKE Ø, 255
```

   Answer: _____
   Will the TRS-80 do as it has been told to do? _____
9. Write a statement to POKE the number 123 into location 17129.
   Answer: _____

10. Screen positions 0 (upper left corner) through 1023 (lower right corner) correspond, one to one, to RAM memory locations 15360 through 16383.

Screen Position	RAM Memory Location
0	15360
1	15361
2	15362
•	•
•	•
•	•
1023	16383

Complete the following equations, relating Screen Positions and RAM Memory Locations.

RAM Location = Screen Position + _____
Screen Position = RAM Location − _____

Answers to Self-Test

1. 0 through 12287
2. PRINT the decimal number contained in memory location 1234.
3. Assign the decimal number contained in memory location 1234 to the variable X.
4. One way to do it. (The READY prompt will follow the display.)

```
100 CLS
110 INPUT "START ADDRESS"; S
120 INPUT "END ADDRESS"; E
130 CLS
140 A$= "LOC & BYTE"
150 PRINT A$, A$, A$, A$
160 FOR X = S TO E
170 PRINT X; PEEK(X),
180 NEXT X
```

5. 12288
6. 17129
7. 4K TRS-80 20479
   16K TRS-80 32767
   32K TRS-80 49151
   48K TRS-80 65535
8. Place the value 255 into memory location 0.
   No. (You cannot POKE into ROM.)
9. POKE 17129, 123
10. +15360
    −15360

CHAPTER THREE

Graphics and Supergraphics

One of the most useful and entertaining features of the TRS-80 is its graphics capabilities. There are several ways to produce useful shapes and moving displays. We will discuss the following four methods in this chapter.

(1) Setting individual, tiny rectangles.
(2) Setting blocks of six rectangles with POKE.
(3) Setting blocks of six rectangles with CHR$.
(4) Setting strings of blocks of six rectangles.

In this chapter you will also learn:

- the difference between video print positions and graphic positions,
- to use SET and RESET to plot graphic rectangles,
- to POKE graphic characters into video screen and memory,
- to display all the graphic characters with the graphic codes that produce them,
- to paint the screen white and POKE holes in the printing by four different methods:

  SET, RESET
  POKE
  CHR$
  STRING$

- to create different shapes and move them across the screen by three different methods:

  POKE
  CHR$
  STRING$

- to construct simple and complex mandalas with the CHR$ function.

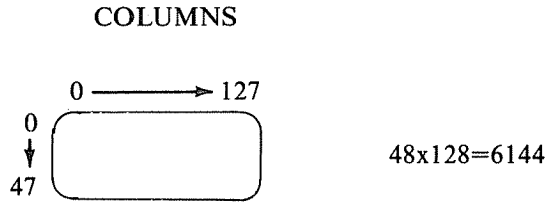
Individual, Tiny Rectangles

The TRS-80 video screen lives two distinct lives.

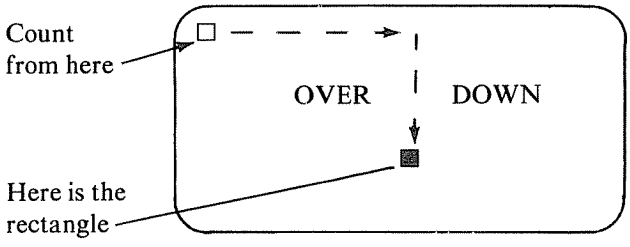
* For text, it has 1024 printing positions, numbered 0 through 1023.
* For graphics, it has 6144 positions, each of which can be occupied by a tiny rectangle of light, much smaller than a "character."

In this chapter, we will discuss both modes and the relationship of one to the other. We begin with the graphics mode, most useful for drawing pictures, graphs, and assorted shapes.

The video screen is divided into 6144 tiny rectangles for display. Each tiny rectangle can be turned on by the SET statement or turned off by the RESET statement. The screen is numbered from the upper left corner. There are 48 rows (numbered 0 through 47) with 128 rectangles in each row (numbered 0 through 127).



Two values must be specified in the SET and RESET statements. The first tells how far OVER from the left side to place the given rectangle. The second value tells how far DOWN from the top to place the rectangle. A comma is put between the two values; this tells the TRS-80 that there are *two* values; one for OVER — the second for DOWN.



The statement is: `SET (OVER, DOWN)`

comma

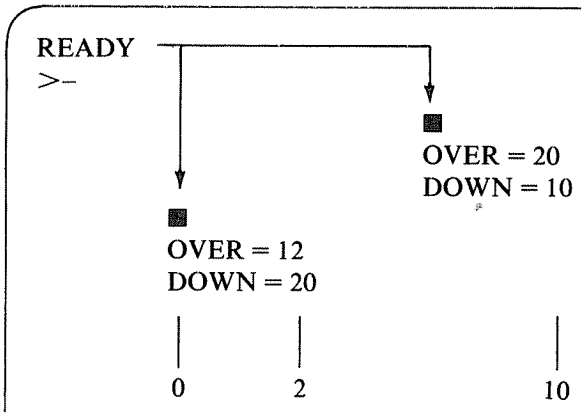
parentheses

Examples:

```
SET (2Ø, 1Ø)
SET (12, 2Ø)
```

Do it like this: `CLS : SET (2Ø, 1Ø) : SET (12, 2Ø)`

And this is what you will see. (Well, of course you won't see our arrows explaining what is happening.)

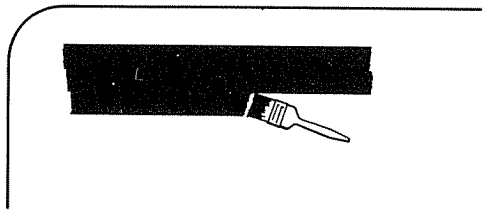


If you wanted to turn on every tiny rectangle on the screen using the SET statement, you could do it like this:

```
1Ø REM *PAINT HORIZONTAL LINES*
2Ø CLS
3Ø FOR DO = Ø TO 47
4Ø   FOR OV = Ø TO 127
5Ø     SET (OV, DO)
6Ø   NEXT OV
7Ø NEXT DO
8Ø GOTO 8Ø
```

This gives 48 rows
128 rectangles in each row
Turn a point on
Next rectangle in row
Next row

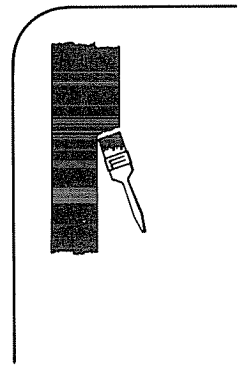
Each time through the inner FOR-NEXT loop, one complete row of rectangles is turned on, and each time through the outer loop, a new row is set up. A white screen is painted with horizontal brush strokes beginning near the top of the screen.



The screen can be painted with vertical stripes by interchanging lines 30 and 40 and also lines 60 and 70.

```
10 REM *PAINT VERTICAL LINES*
20 CLS
30 FOR OV = 0 TO 127
40   FOR DO = 0 TO 47
50     SET (OV, DO)
60   NEXT DO
70 NEXT OV
80 GOTO 80
```

Now the screen is painted with top to bottom vertical stripes, beginning near the left edge of the screen.



Now let's use the white screen, but randomly erase some of the points on it. You'll need a random value for OVer, between the values 0 and 127, inclusive. DOwn will range from 0 to 47 inclusive.

```
 10 REM *PAINT HORIXONTAL LINES*
 20 CLS
 30 FOR DO = 0 TO 47
 40   FOR OV = 0 TO 127
 50     SET (OV, DO)
 60   NEXT OV
 70 NEXT DO
 80 REM *ERASE SOME POINTS*
 90 DO = RND (48)-1 <---- Gives a random value (0 through 47)
100 OV = RND (128)-1 <--- Gives a random value (0 through 127)
110 RESET (OV, DO) <----- Erases a point
120 GOTO 90
```

The screen is painted white, and strange patterns appearing as tiny rectangles are erased. Eventually, it begins to look like a maze of black dots on a white background. Then it slowly transforms into white dots on a black background as more and

more points are erased. Look for interesting patterns along the way. When you tire of watching, press the BREAK key to stop the program.

You can restrict the point erasures to a certain part of the screen in a variety of ways. Here are two variations.

1. Change line 90 to:

```
9Ø DO = RND (24) -1
```

(top half of screen)

2. Change line 100 to:

```
1ØØ OV = RND (64)-1
```

(left half of screen)

Your turn. Show how to do each of the following variations.

3. Bottom half of screen. Change line 90 to:

_____

4. Right half of screen. Change line 100 to:

_____

5. Center portion of screen. Change lines 90 and 100 to:

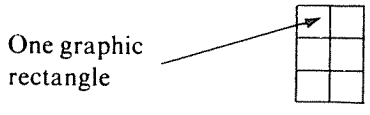
_____

_____

```
3.  9Ø DO = 48 - RND(24)
4. 1ØØ OV = 128 - RND (64)
5.  9Ø DO = 36 - RND (12)
   1ØØ OV = 96 - RND (32)
```

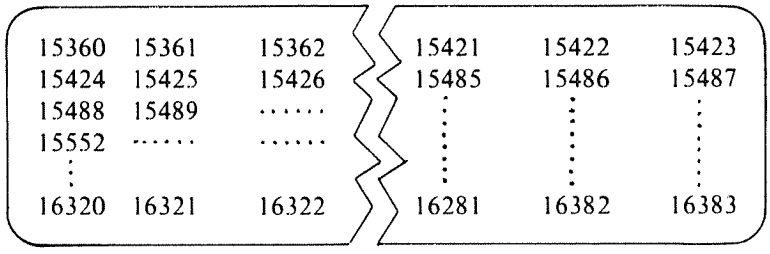
Setting Blocks of Six Rectangles with POKE

Each character printed on the screen occupies the same space as 6 of the tiny graphic rectangles.



One PRINT position
(=6 graphic rectangles)

You know from the PRINT@ statement that text can be printed at any one of 1024 print positions (numbered from 0 through 1023, inclusive). Each print position occupies one storage location in the memory used for the video display. The map of the video memory locations is like this:



Remember, each memory block contains 6 rectangular graphic cells. The relationship between memory addresses and the row and column numbers used in SET and RESET statements is indicated in the following diagram:

Memory Location 15360

OV=0 DO=0	OV=1 DO=0
OV=0 DO=1	OV=1 DO=1
OV=0 DO=2	OV=1 DO=2

Memory Location 15361

OV=2 DO=0	OV=3 DO=0
OV=2 DO=1	OV=3 DO=1
OV=2 DO=2	OV=3 DO=2

Memory Location 15362

OV=4 DO=0	OV=5 DO=0
OV=4 DO=1	OV=5 DO=1
OV=4 DO=2	OV=5 DO=2

You can POKE data into any one of the video memory locations by using the POKE statement. If the correct data is POKEd into a given location, a pattern composed of a combination of these 6 rectangles will turn on. The data, if between 128 and 191, produces a pattern called a graphic character.

WARNING!!! AVOID POKING DATA INTO THE WRONG ADDRESS

The video addresses for the graphic characters must be in the range of 15360 through 16383, inclusive.

A table of graphic characters is given in appendix A. However, it is easy to construct a graphic character from the following:

1. Number each of the 6 cells with a power of 2 in the order shown.

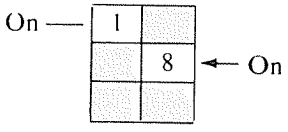
1	2
4	8
16	32

or

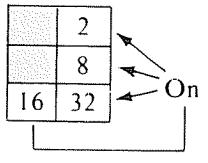
2^0	2^1
2^2	2^3
2^4	2^5

2. The number 128 creates a blank character (all tiny rectangles OFF).
3. Decide what rectangles you want turned on.
4. The data used to form your graphic character is the sum of 128 and the value(s) in the cell(s) that you want to light.

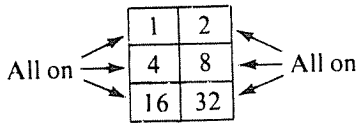
Examples:



Add 128
8
+ 1
137 is the graphic character



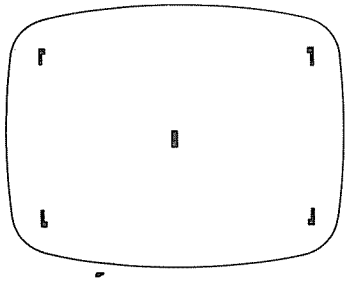
Add 128
32
16
8
+ 2
186 is the graphic character



Add	128
	32
	16
	8
	4
	2
	+ 1
	191

is the graphic character

To put a graphics character on the screen, you must put together the correct combination of video memory location and graphic code and POKE the code into the memory. Suppose you want to display the following:



			Graphic Code	Memory Location
Upper left	128+1+2+4	=	135	15360
Upper right	128+1+2+8	=	139	15423
Lower left	128+4+16+32	=	180	16320
Lower right	128+8+16+32	=	184	16383

This short program would do the job.

```
10 CLS
20 POKE 15360,135: POKE 15423,139
30 POKE 16320,180: POKE 16383,184
```

To see the complete table of graphic character codes and their shapes, enter and run this program:

Display Graphic Characters

```
  1Ø REM *INITIALIZE*
  2Ø CLS
  3Ø M=15360: A=129

 1ØØ REM *PRINT A ROW OF GRAPHICS*
 11Ø FOR C= A to A+3
 12Ø   IF C> 191 GOTO 3ØØ
 13Ø   PRINT C,; POKE M, C
 14Ø   M = M+16
 15Ø   IF M>16383 GOTO 3ØØ
 16Ø NEXT C

 2ØØ REM *READY NEXT ROW*
 21Ø IF A = 149 or A=173 GOSUB 1ØØØ
 22Ø A = A+4
 23Ø M = M+64
 24Ø IF M> 16383 GOTO 3ØØ
 25Ø PRINT
 26Ø GOTO 1ØØ

 3ØØ REM *ALL DONE*
 31Ø GOTO 31Ø

1ØØØ REM *END OF SCREEN BLOCK*
1Ø1Ø PRINT: PRINT
1Ø2Ø INPUT ''PRESS ENTER TO CONTINUE''; A$
1Ø3Ø M=15360
1Ø4Ø CLS
1Ø5Ø RETURN
```

Don't go beyond code 191

Print code and character

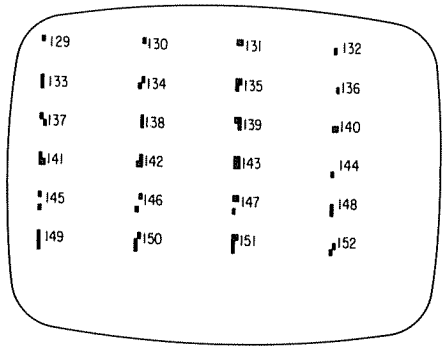
Don't go beyond screen memory

Test for full screen

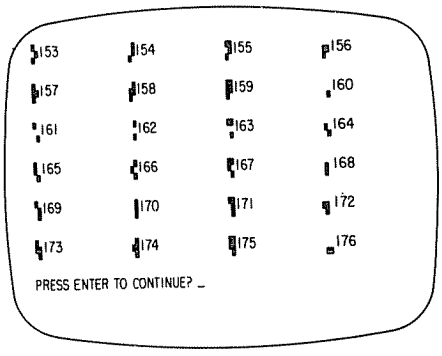
Double check

Go get another row

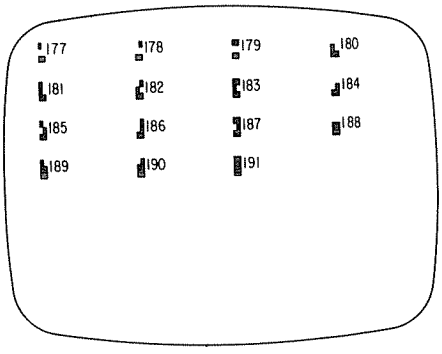
- Type RUN and press ENTER. This is what you see:



- Press ENTER and this happens:



- Press ENTER again to get the last bunch of graphics characters:



The program works like this:

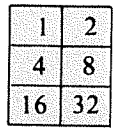
Lines 10 through 30 initialize the first graphic character and the first screen memory position.

Lines 100 through 160 print a line of graphic codes and their corresponding characters (4 of each on a line).

Line 210 tests to see if the screen is full. If so, the subroutine at line 1000 is executed. You can study the screen of characters as long as you want. When you press a key, the subroutine initializes the screen again and proceeds with the next screen of characters. If the screen is not full, line 260 returns the program to line 100 for another line of characters.

Notice that another safety check is made at line 240 to be sure that the memory location where data is to be POKEd is not beyond the boundary 16383. Line 250 provides a space between lines for easier viewing.

Do you remember how slowly the screen was painted white by the SET statement? The graphics code 191 produces a complete block of 6 rectangles.



128+1+2+4+8+16+32 = 191
All six rectangles on

Compare the results of the program that painted the screen white using the SET statement with the following program that POKEs the screen white with graphic code 191.

```
1Ø REM *LIGHT A WHOLE BLOCK*
2Ø CLS
3Ø FOR M = 1536Ø TO 16383
4Ø   POKE M,191
5Ø NEXT M
6Ø GOTO 6Ø
```

Fill every screen memory location with code 191.

This program takes about seven seconds to run compared to forty-seven seconds for the program using the SET statement.

Now see how fast black holes can be put in with POKE statements. Add these lines to lines 10 through 50.

```
6Ø REM *POKE BLACK HOLES*
7Ø R = RND(1Ø24)+15359
8Ø POKE R,128
9Ø GOTO 7Ø
```

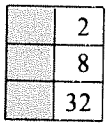
A blank block of 6 rectangles

The holes that get POKEd into the white screen are much larger. Therefore, the transition from black on white to white on black takes place much faster.

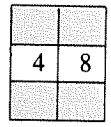
Let's move on to something a little more entertaining. Graphic codes can be POKEd in to produce a three-car dragster race across the video screen. Work with only one car in the design of the program first. Break the program into these four steps:

1. Decide on the graphics for the car.
2. Decide how to make the car move.
3. Add two more cars.
4. Determine the winner.

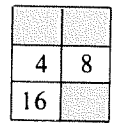
First, to keep the car's design as simple as possible, limit it to three graphics blocks.



BLOCK1



BLOCK2



BLOCK3

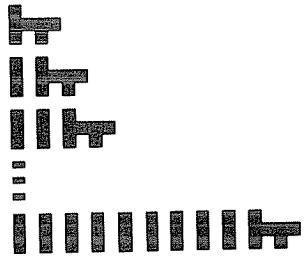
Block 1 = 128+2+8+32 = 170
Block 2 = 128+4+8 = 140
Block 3 = 128+4+8+16 = 156

This may not look like much of a dragster, but with a little imagination it will do.

The BASIC statements to draw the dragster are:

Block 3: POKE M+2, 156
Block 2: POKE M+1, 140
Block 1: POKE M, 170

Second, move the dragster by poking each block one place to the right each time, using a FOR-NEXT loop. Follow the car with a blank space to erase the last block or else the dragster will leave tracks across the screen. WATCH MY DUST!



etc.

The FOR-NEXT loop looks like this:

```
FOR M = 15360 TO 15420
 POKE M+3, 156: POKE M+2, 140: POKE M+1, 170: POKE M, 128
NEXT M
```

A blank space

Before going to step three, put the first two steps together and try it out.

```
10 CLS
20 FOR M = 15360 TO 15420
30   POKE M+3,156: POKE M+2,140: POKE M+1, 170: POKE M,
40 NEXT M
50 GOTO 50
```

Across the screen it goes from start to finish. How would it do with some competition? Let's find out.

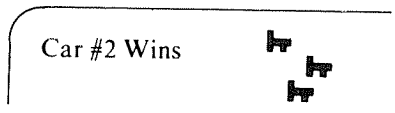
Third, set up three cars to speed across the screen. Of course you must add some element of chance. Add a random value for movement to give each car a chance to win.

Three Car Dragster Race

```
 1Ø REM *POKE CAR RACE*
 2Ø CLS
 3Ø A=Ø; B=Ø: C=Ø
1ØØ REM *RACE ON*
11Ø FOR M = 15359 TO 15423
12Ø   A=A+RND(2)-1: B=B+RND(2)-1:C=C+RND(2)-1
13Ø   D=M+A: E=M+B+128: F=M+C+256
14Ø   POKE D+3,156: POKE D+2,14Ø: POKE D+1,17Ø:POKE D,128:
      POKE D-1,128
15Ø   POKE E+3, 156: POKE E+2,14Ø: POKE E+1,17Ø:POKE E,128
      POKE E-1,128
16Ø   POKE F+3,156:POKE F+2,14Ø:POKE F+1,17Ø:POKE F,128
      POKE F-1,128
17Ø   IF D+3>15421 OR E+3>15549 OR F+3>15677 THEN 21Ø
18Ø NEXT M
2ØØ REM *PICK WINNER*
21Ø IF A>=B AND A>=C THEN D=1 :GOTO 24Ø
22Ø IF B>=C THEN D=2:GOTO 24Ø
23Ø D=3
24Ø PRINT @54Ø,"CAR #"D"WINS"
```

The three cars move forward either one or two spaces with each pass through the loop. The increase in the value of M accounts for one space. The other space is provided by line 120, where A, B, and C are increased by a value of either 1 or 0, depending on the random number selected. Line 170 serves as a check to stop the race when one car crosses the finish line.

Fourth, lines 210 through 230 determine which car wins the race. There is a slight bias for A in case of a tie with B and/or C, and a slight bias for B in case of a tie with C. A typical end of the race might look like this:



Since the POKE command can set 6 rectangles at once, the race is much more realistic than it would be by SETting each individual rectangle. Notice the two POKE 128 statements at the end of each car. These erase the back of the car as it moves forward. Two blank spaces must now follow each car since the car can now move two spaces forward as the FOR-NEXT loop is executed.

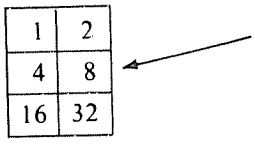
Setting Blocks of Six Rectangles with CHR$

The CHR$ function returns a one-character string whose ASCII code, control, or graphics character is specified. Using it for graphics, the form is:

```
PRINT CHR$ (exp)
           ↑
           └── a graphics code, 128 through 191
```

Since you will now be printing the characters instead of poking them into memory, you must use print positions instead of actual video memory locations to display the graphic characters where you want them. To print a full 6-block graphics character:

```
PRINT CHR$ (191)
```



128+1+2+4+8+16+32 = 191

Go back and paint the screen white using CHR$.

```
1Ø REM *PAINT IT WHITE*
2Ø CLS
3Ø FOR S = Ø TO 1Ø22
4Ø   PRINT @S,CHR$(191);
5Ø NEXT S
6Ø POKE 16383,191
7Ø GOTO 7Ø
```

Line 60 reverted to the POKE statement to avoid scrolling the screen. Whenever PRINT @1023 is executed, the screen automatically scrolls one line. We don't want that to happen now so we use a POKE statement to paint screen position 1023 white.

This looks very similar to the program on page 22. Try it. Then punch some black holes in the white screen by adding these lines:

```
7Ø REM *PRINT BLACK HOLES*
8Ø S = RND (1Ø24)-1
9Ø PRINT @S, CHR$ (128)
1ØØ GOTO 8Ø
```

← a blank space (black hole)

1. Try the addition. Which is faster, POKE or PRINT CHR$? _____

---

1. They both take about the same time. (They should, since each does the same thing — puts a graphic character into successive display positions. They simply use a different BASIC statement to accomplish the same thing.)

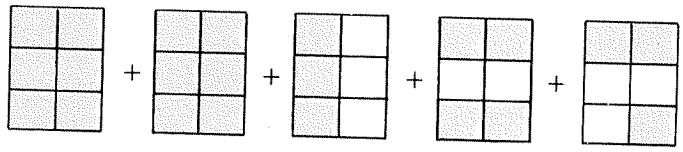
Now, change the racing car program on page 47 to use CHR$ instead of POKE. As you do so, remember that strings can be concatenated (joined together).

Example:

```
PRINT @D, CHR$(128)+CHR$(128)+CHR$(17Ø)+CHR$(14Ø)+CHR$(156)
```

or

```
A$=CHR$(128)+CHR$(128)+CHR$(17Ø)+CHR$(14Ø)+CHR$(156)
PRINT @D, A$
```



The following is one way to alter the three-car dragster race to use CHR$:

changes:

```
11Ø FOR M = Ø TO 6Ø

14Ø PRINT @D, A$
15Ø PRINT @E, A$
16Ø PRINT @F, A$
17Ø IF D+3>60 OR E+3>188 OR F+3>316 THEN 21Ø
```

addition:

```
 4Ø A$=CHR$(128)+CHR$(128)+CHR$(17Ø)+
      CHR$(14Ø)+CHR$(156)
```

This makes the dragsters move across the screen faster. The dragster shape is generated *once* at line 40. In the original program, the shape was generated three times on every pass through the loop. Therefore, this calculation time is greatly reduced.

Three-Car Dragsters with CHR$

```
 1Ø REM *CHR$ CAR RACE*
 2Ø CLS
 3Ø A=Ø: B=Ø: C=Ø
 4Ø A$=CHR$(128)+CHR$(128)+CHR$(17Ø)+CHR$(14Ø)+CHR$(156)
1ØØ REM *RACE ON*
11Ø FOR M = Ø TO 6Ø
12Ø   A=A+RND(2)-1: B=B+RND(2)-1: C=C+RND(2)-1
13Ø   D=M+A: E=M+B+128: F=M+C+256
14Ø   PRINT @D,A$
15Ø   PRINT @E,A$
16Ø   PRINT @F,A$
17Ø   IF D+3>6Ø OR E+3>188 OR F+3>316 THEN 21Ø
18Ø NEXT M
2ØØ REM *PICK A WINNER*
21Ø IF A>=B AND A>=C THEN D=1 : GOTO 24Ø
22Ø IF B>=C THEN D=2: GOTO 24Ø
23Ø D=3
24Ø PRINT @54Ø, "CAR # "D" WINS"
```

Mandalas

A mandala is a pattern used in meditation. It is usually symmetrical left, right, up, and down. The next three programs illustrate methods to create simple and complex mandalas.

Start by printing a solid block near the center of the screen. Then, print one graphic code in each corner (upper left, upper right, lower left, and lower right). Assign the variables A, B, C, D, E, and F to the rectangles for the upper left corner. The computer will generate random 1's or 0's for each of the six variables (1 for ON and 0 for OFF).

A	B
C	D
E	F

or

2^0	2^1
2^2	2^3
2^4	2^5

for upper left

The value for the graphic block in the upper left corner is then:

$$UL = 128+A+2*B+(2*C)\uparrow 2+(2*D)\uparrow 3+(2*E)\uparrow 4+(2*F)\uparrow 5$$

Example:

If A and D = 1 and all the rest = 0,

$$UL = 128+1+0+0+(2*1)\uparrow 3+0+0 = 128+1+8 = 137$$

Then PRINT CHR$(137) in the upper left corner.

To make the complete picture symmetrical, you can define the other rectangles as follows:

Upper left

A	B
C	D
E	F

Upper right

B	A
D	C
F	E

Center
CHR$(191)

E	F
C	D
A	B

Lower left

F	E
D	C
B	A

Lower right

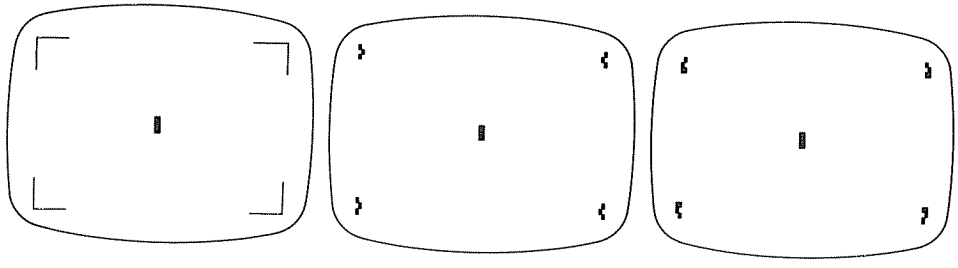
This makes the complete picture symmetrical left, right, up, and down.

Center and Four-Corner Mandala

```
 1Ø REM *SET UP CHARACTER CODES*
 2Ø CLS
 4Ø A=RND(2)-1:B=RND(2)-1:C=RND(2)-1:D=RND(2)-1:E=RND(2)-
    F=RND(2)-1
1ØØ UL=128+A+2*B+(2*C)↑2+(2*D)↑3+(2*E)↑4+(2*F)↑5
11Ø UR=128+B+2*A+(2*D)↑2+(2*C)↑3+(2*F)↑4+(2*E)↑5
12Ø LL=128+E+2*F+(2*C)↑2+(2*D)↑3+(2*A)↑4+(2*B)↑5
13Ø LR=128+F+2*E+(2*D)↑2+(2*C)↑3+(2*B)↑4+(2*A)↑5
2ØØ REM *PRINT THE MANDALA*
21Ø PRINT @544,CHR$(191);:PRINT @144,CHR$(UL);
22Ø PRINT @176,CHR$(UR);:PRINT @912,CHR$(LL);
23Ø PRINT @944,CHR$(LR);
3ØØ GOTO 1Ø
```

Note: Powers of 2

Line 40 creates random 1's or 0's for the variables A, B, C, D, E, and F. Lines 100 through 130 use these variables to create symmetrical graphic codes for upper left (UL), upper right (UR), lower left (LL), and lower right (LR). Lines 210 through 220 print the graphics at their respective positions. The program then repeats, giving a different pattern at the corners each time.



■ Enter the program and let it run for awhile before going to the next program. Don't erase the program; you'll add to it later.

By making a few additions and changes, you can extend the pattern inward from the four corners toward the center to form a large X shape.

Add:

```
 3Ø FOR N = Ø TO 33Ø STEP 66
 5Ø   IF N=Ø THEN M=Ø ELSE M=62*(N/66)

24Ø NEXT N
25Ø FOR WAIT = 1 TO 2ØØ: NEXT WAIT
```

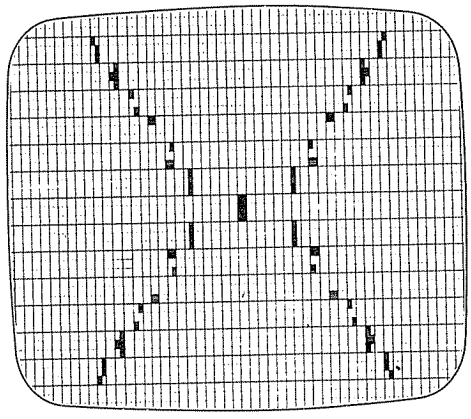
Change:

```
21Ø PRINT @544, CHR$(191);: PRINT @144+N,CHR$(UL);
22Ø PRINT@176+M,CHR$(UR);: PRINT @912-M,CHR$(LL);
23Ø PRINT @944-N,CHR$(LR);
```

The altered program:

```
 1Ø REM *SET UP CHARACTER CODES*
 2Ø CLS
 3Ø FOR N = Ø TO 33Ø STEP 66
 4Ø   A=RND(2)-1:B=RND(2)-1:C=RND(2)-1:D=RND(2)-1:E=RND(2)-1:
      F=RND(2)-1
 5Ø   IF N=Ø THEN M=Ø ELSE M=62*(N/66)
1ØØ   UL=128+A+2*B+(2*C)↑2+(2*D)↑3+(2*E)↑4+(2*F)↑5
11Ø   UR=128+B+2*A+(2*D)↑2+(2*C)↑3+(2*F)↑4+(2*E)↑5
12Ø   LL=128+E+2*F+(2*C)↑2+(2*D)↑3+(2*A)↑4+(2*B)↑5
13Ø   LR=128+F+2*E+(2*D)↑2+(2*C)↑3+(2*B)↑4+(2*A)↑5
2ØØ   REM *PRINT THE MANDALA*
21Ø   PRINT @544,CHR$(191);: PRINT @144+N,CHR$(UL);
22Ø   PRINT @176+M,CHR$(UR);: PRINT @912-M,CHR$(LL);
23Ø   PRINT @944-N,CHR$(LR);
24Ø NEXT N
25Ø FOR WAIT = 1 to 2ØØ: NEXT WAIT
3ØØ GOTO 1Ø
```

■ Enter and let the program run awhile. Here is a possible shape:



After a brief time delay, the screen is erased and a new pattern is drawn.

Enter and run this program. If you wish, eliminate the time delay at line 250 to create a continually changing pattern. Do *not* erase the program.

■ After you tire of the X-shaped mandala, make the following additions and changes.

Add:

```
 25 S=1Ø
 35 FOR R= Ø TO S

235 NEXT R
237 S = S-1
```

Change:

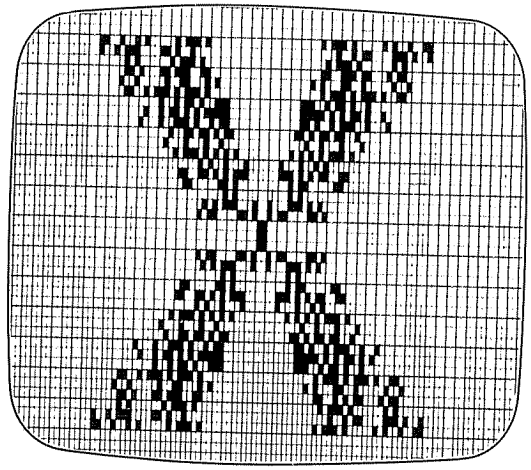
```
21Ø PRINT @544,CHR$(191);: PRINT @144+N+R,CHR$(UL);
22Ø PRINT @176+M-R,CHR$(UR);:PRINT @912-M+R,CHR$(LL);
23Ø PRINT @944-N-R,CHR$(LR);
```

Fancy Mandala

```
 1Ø REM *SET UP CHARACTER CODES*
 2Ø CLS
 25 S= 1Ø
 3Ø FOR N =Ø TO 33Ø STEP 66
 35   FOR R = Ø TO S
 4Ø     A=RND(2)-1:B=RND(2)- 1:C=RND(2) -1:D =RND(2)-1:
        E=RND(2)-1:
        F=RND(2)-1
 5Ø     IF N=Ø THEN M=Ø ELSE  M=62*(N/66)
1ØØ     UL=128+A+2*B=(2*C)↑2+(2*D)↑3+(2*E)↑4+(2*F)↑5
11Ø     UR=128+B+2*A+(2*D)↑2 +(2*C)↑3+(2*F)↑4+(2*E)↑5
12Ø     LL=128+E+2*F+(2*C)↑2 +(2*D)↑3+(2*A)↑4+(2*B)↑5
13Ø     LR=128+F+2*E+(2*D)↑2+(2*C)↑3+(2*B)↑4+(2*A)↑5
2ØØ     REM *PRINT THE MANDALA*
21Ø     PRINT @544,CHR$(191);:PRINT@144+N+R,CHR$(UL);
22Ø     PRINT @176+M-R,CHR$(UR);:  PRINT@912-M+R,CHR$(LL);
23Ø     PRINT @944-N-R,CHR$(LR);
235    NEXT R
237    S = S-1
24Ø NEXT N
25Ø FOR WAIT = 1 TO 2ØØ: NEXT WAIT
3ØØ GOTO 1Ø
```

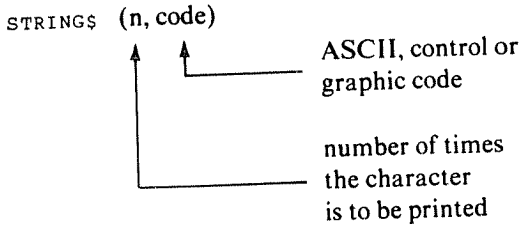
■ Enter the program and let it run. It is rather slow, but it lets you see the mandala grow from the outside inward. Enjoy it awhile before going on.

Here is a sample mandala:



Setting Strings of Graphic Blocks

When more than one graphic block having the same code is to be displayed, the STRING$ function can be used. The format for this function is:



■ Returning once again to painting a white screen, use the following program:

```
10 REM *PAINT IT WHITE*
20 CLEAR 100
30 CLS
40 FOR S = 0 to 896 STEP 64
50   PRINT @S,STRING$(64, 191)
60 NEXT S
70 PRINT @960, STRING$(63,191)
80 POKE 16383,191
90 GOTO 90
```

Enlarge the string space

A full line of blocks

To avoid scrolling, only 63 blocks in this line

Comparing the time to completely paint the screen, this program takes about one second versus seven seconds for POKE and CHR$ and forty-seven seconds for SET.

■ To punch in a random sized block of holes, change line 90 and add the lines as shown:

```
 90 REM *PUNCH BLACK HOLES*
100 M = RND(32)
110 P = RND(991)
120 PRINT @P, STRING$(M,128);
130 GOTO 100
```

Random length block (1 through 32)

Random address

This addition can blank out up to half a line of the screen at a time. The random value selected for M can be changed to provide shorter or longer length blocks. This can really black out the screen in a hurry.

After you have tried the painting and hole punching program, return to the three-car dragster program. This time use the STRING$ function along with CHR$ to print each car's progress. Make each car appear to move by increasing the number of blank spaces following it.

The car is denoted by:

```
A$ = CHR$(170)+CHR$(140)+CHR$(156)
```

To move the car along, use an ever increasing number of blank spaces by the STRING$ function.

```
PRINT @Ø,STRING$(D,191)+A$
```

Where D increases for each printing.

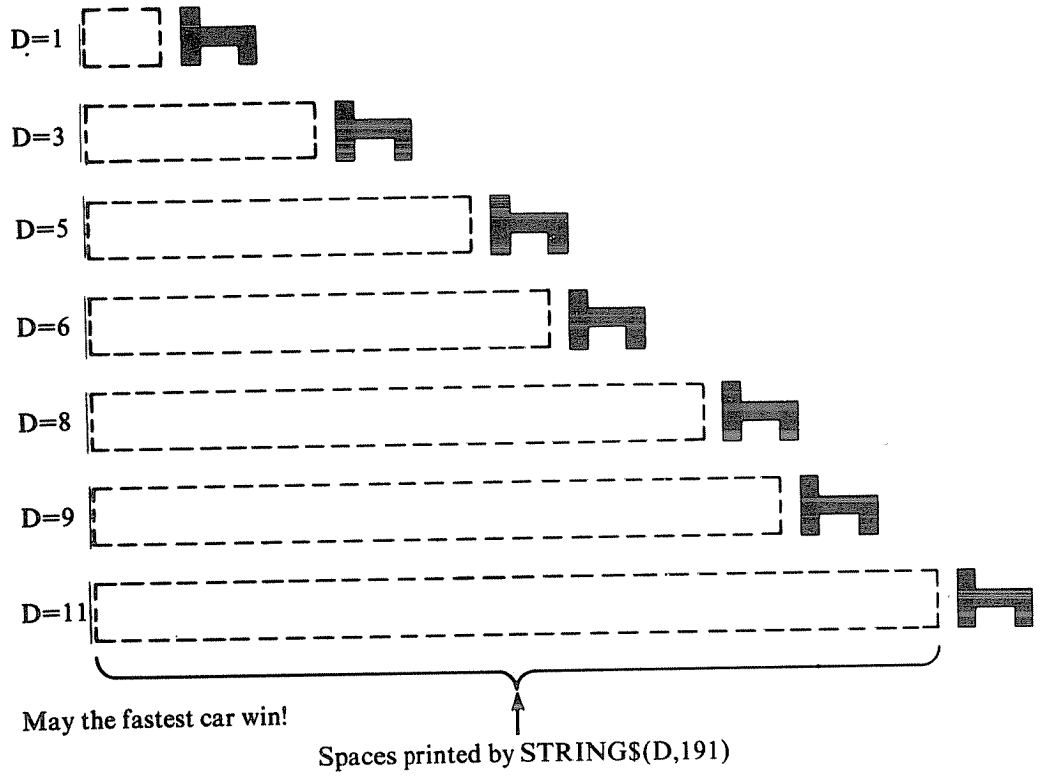
Three-Car Dragsters with Strings

```
 1Ø REM *STRING$ CAR RACE*
 2Ø CLEAR 2ØØ
 3Ø CLS:A=Ø:B=Ø:C=Ø
 4Ø A$ = CHR$(17Ø)+CHR$(14Ø)+CHR$(156)
1ØØ REM *RACE ON*
11Ø FOR M = Ø TO 6Ø
12Ø   A=A+RND(2)-1: B=B+RND(2)-1:C=C+RND(2)-1
13Ø   D=M+A: E=M+B: F=M+C
14Ø   PRINT @Ø,STRING$(D,191)+A$
15Ø   PRINT @128,STRING$(E,191)+A$
16Ø   PRINT @256,STRING$(F,191)+A$
17Ø   IF D 58> OR E >58 OR F> 58 THEN 21Ø
18Ø NEXT M
2ØØ REM *PICK A WINNER*
21Ø IF A>=B AND A>=C THEN D=1: GOTO 24Ø
22Ø IF B>=C THEN D=2: GOTO 24Ø
23Ø D=3
24Ø PRINT @54Ø, "CAR #"D"WINS"
```

The results of this race look much the same as before; however, the printing technique is quite different. During each pass through the FOR-NEXT loop, each car is printed on a separate line, but the length of blank spaces to the left of the cars increases each time. Here is a diagram of successive printings for one car.

Example: Suppose D increases each time through the loop in the following manner:

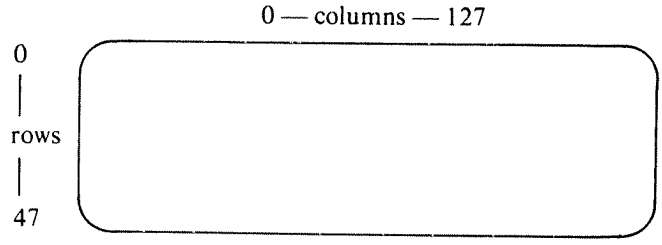
D=1,3,5,6,8,9,11,etc.



Summary

If you weren't a graphics expert before this chapter, you should be by now. You have learned:

- The video screen consists of 48 rows with 128 rectangles in each row that can be individually SET or RESET.



- A block of 6 (2 columns by 3 rows) rectangles can be POKEd into video screen memory. Each pattern of 6 rectangles is specified by a character code. The block of 6 rectangles corresponds to a PRINT position.
- The graphics codes (128, all rectangles off through 191, or all rectangles on) can be POKEd into the video memory associated with any PRINT position. Print positions are 0 through 1023. The corresponding video memory locations are 15360 through 16383.
- To use the PRINT position graphics codes with the CHR$ function. Since this method PRINTs the graphic characters instead of poking them into memory, the print positions can be specified directly.
- To set a whole string of graphic character blocks with the STRING$ function. This function allows you to specify the graphics code and how many times it is to be used successively.
- To use sample demonstrations of all four graphics display methods:

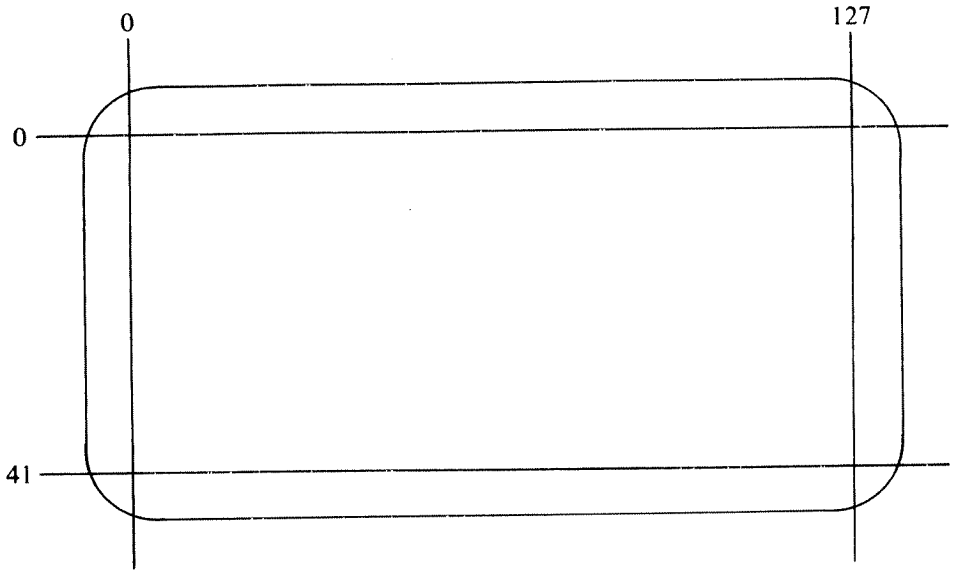
  SET, RESET
  POKE
  CHR$
  STRING$

Self-Test

1. Before you RUN the following program, draw a sketch of the screen showing what it will look like after the TRS-80 has reached line 80.

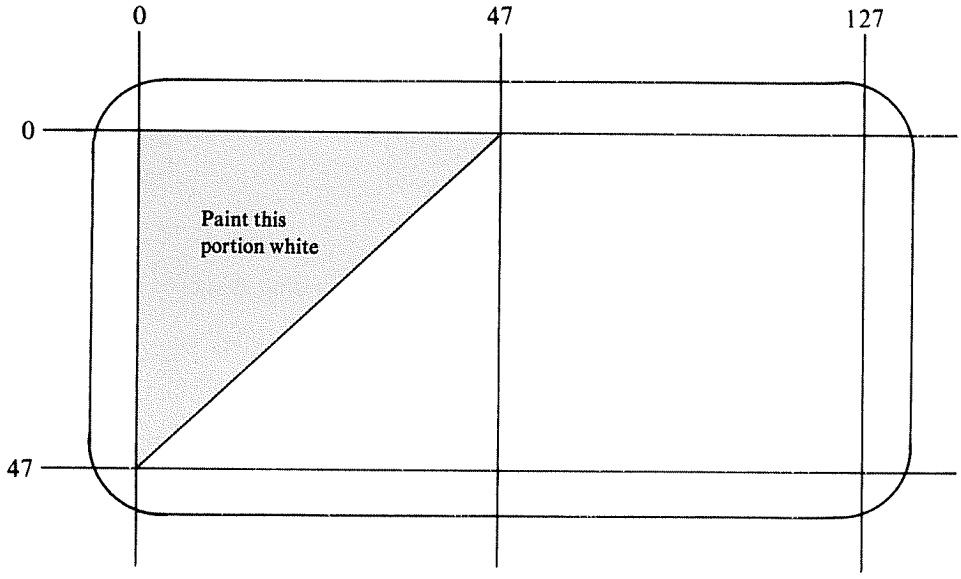
```
10 REM**PAINT PART OF THE SCREEN WHITE
20 CLS
30 FOR DO = 0 TO 47
40   FOR OV = 0 TO DO
50     SET (OV,DO)
60   NEXT OV
70 NEXT DO
80 GOTO 80
```

Draw your sketch *before* you RUN the program!

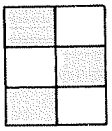


2. In exercise 1, the screen is painted with horizontal stripes, beginning at the top of the screen. Write a program to produce exactly the same effect using vertical stripes, beginning at the left edge of the screen.

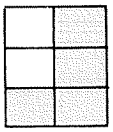
3. Write a program to paint the screen as shown in the following sketch.



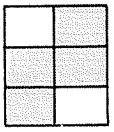
4. For each graphics shape, write the corresponding numeric code below the shape. Remember: The code is equal to 128 plus the values of the turned on (■) tiny rectangles.

(a) 

_____

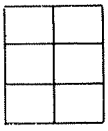
(b) 

_____

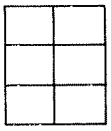
(c) 

_____

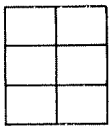
5. For each graphics character code, show the corresponding shape.

(a) 

166

(b) 

150

(c) 

185

6. Write statements to:

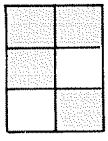
(a) POKE the graphics code 185 into video memory location 16000:

_____

What screen position is this?_____

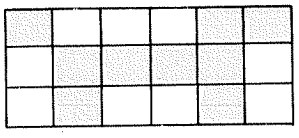
(b) POKE the graphics code 185 into screen position 1000:

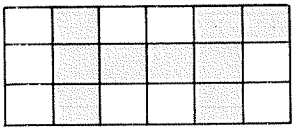
_____

(c) POKE the graphics code for  into screen position 5000:

_____

7. Below are two sketches of a "dog." Each consists of three graphics characters.





The sketches are the same, except for the position of the dog's "tail." Write a program to show the dog "wagging" its tail. Put the dog near the center of the screen, perhaps in print positions 543, 544, and 545.

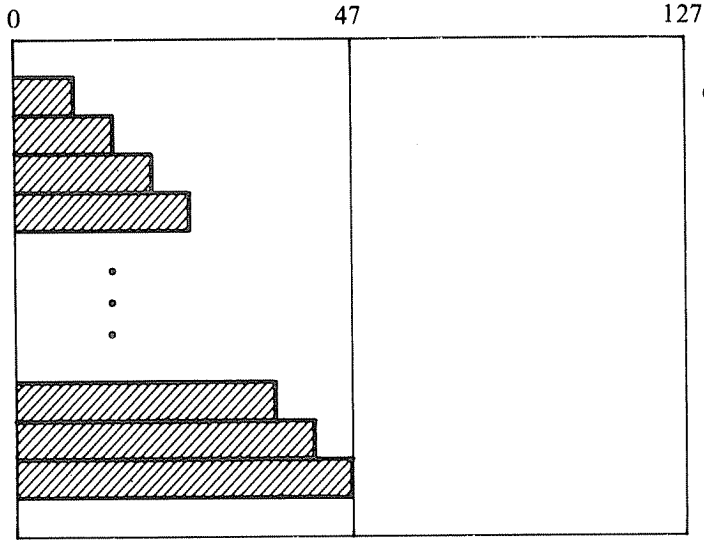
8. Write a program to move your happy dog of exercise 7 across the screen from screen position 512 past the right edge of the screen then, after a brief pause, repeat. Of course, your dog's tail should wag as he goes joyfully across the screen.

   (a) First, write the program using POKE statements.

(b) Then, rewrite the program, using the STRING$ function to move the dog.

Answers to Self-Test

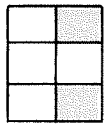
1.

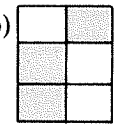


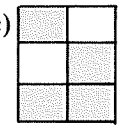
2.
```
1Ø REM ** PAINT PART OF THE SCREEN WHITE #2
2Ø CLS
3Ø FOR OV = Ø TO 47
4Ø   FOR DO - OV TO 47
5Ø     SET(OV,DO)
6Ø   NEXT DO
7Ø NEXT OV
8Ø GOTO 8Ø
```

3.
```
1Ø REM ** PAINT PART OF THE SCREEN WHITE#3
2Ø CLS
3Ø FOR OV = Ø TO 47
4Ø   FOR DO = Ø TO 47 - OV
5Ø     SET(OV,DO)
6Ø   NEXT DO
7Ø NEXT OV
8Ø GOTO 8Ø
```

4. (a) 153 (b) 186 (c) 158

5. (a) 
166
(128+32+9+2)

(b) 
150
(128+16+4+2)

(c) 
185
(128+32+16+8+1)

6. (a) POKE 16000, 185
640
(b) POKE 16460, 185
(c) POKE 15860, 167

7. Here is one way to program it.

```
1Ø REM ** WAG THE TAIL
2Ø CLS
3Ø PRINT@545, CHR$(151);
4Ø PRINT@544, CHR$(140);
5Ø PRINT@543, CHR$(169);
6Ø FOR X = 1 TO 2Ø: NEXT X
7Ø PRINT@543, CHR$(17Ø);
8Ø FOR X = 1 TO 2Ø: NEXT X
9Ø GOTO 5Ø
```

8. There are many ways to write programs a and b. We show one possible program for part b.

```
 1Ø REM ** WAG TAIL AND MOVE DOG.
 2Ø CLEAR 1ØØØ
 3Ø CLS
 4Ø A$ = CHR$(169) + CHR$(14Ø) + CHR$(151)
 5Ø B$ = CHR$(17Ø) + CHR$(14Ø) + CHR$(151)
 6Ø FOR P = 512 TO 572 STEP 2
 7Ø   FOR Y = 1 TO $
 8Ø     PRINT@P, STRING$(P-511,128)+B$;
 9Ø     FOR X = 1 TO 1Ø : NEXT X
1ØØ     PRINT@P, STRING$(P-511,128)+A$;
11Ø   NEXT Y
12Ø NEXT P
13Ø CLS
14Ø FOR X = 1 TO 5Ø: NEXT X
15Ø GOTO 6Ø
```

CHAPTER FOUR

Introduction to Cassette Data Files

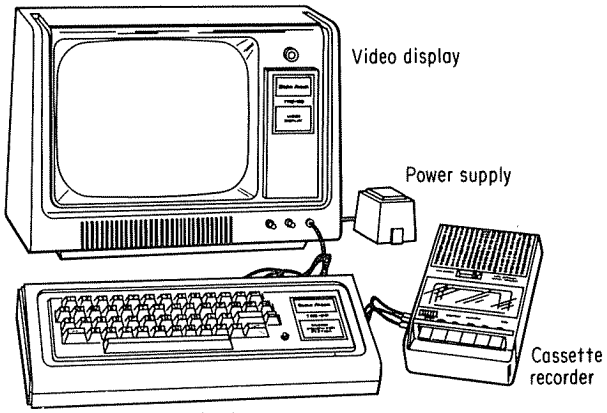
Many computer programs require entering long lists of data that are used by the program. This is often done by using INPUT statements, which takes time and causes lots of wear on finger tips if much data is to be entered. And, if your program is used again, the data must be entered all over again. It's true that READ and DATA statements may be used to include the data in the program, but if new datum is necessary, the program must be rewritten each time the data are changed.

In this chapter you will explore the use of the cassette recorder to enter and save data that may be used again at a later time. You will learn:

- how to handle data (made up of numbers, strings, or a mixture of the two) with the cassette recorder,
- to use the PRINT #-1 statement to transfer data from memory to the cassette recorder,
- to use the INPUT #-1 statement to transfer data from the cassette recorder to the computer's memory,
- what kind of cassettes to buy and how to care for and use them,
- the format used on tapes,
- the difference between a data record and a data file,
- to use demonstration programs to input, save, and retrieve data,
- how to conserve the amount of tape used in saving data, and
- other technical information about cassette recording.

Using Your Recorder

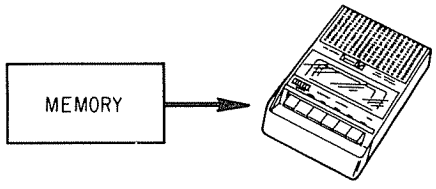
A cassette tape recorder, shown below, is standard equipment with the TRS-80 system.



You have probably used the cassette recorder to load cassette programs into memory (RAM, of course) using the CLOAD command. You may also have used the cassette recorder to CSAVE your own programs on cassette tapes or to make back-up copies of cassette programs you have purchased.*

If you have not used CLOAD and CSAVE, we suggest you read appendix B, "The Cassette Recorder", before continuing with this chapter.

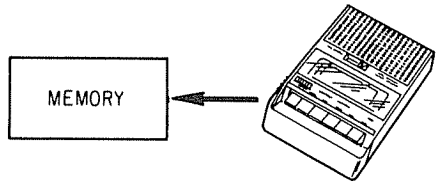
- When you CSAVE a program, the TRS-80 records the program from its memory onto a cassette, using the cassette recorder.
  CSAVE: FROM MEMORY TO TAPE



- When you CLOAD a program, the TRS-80 reads a program from a cassette into its memory, again using the cassette recorder.

*It is OK to make back-up copies of copyrighted software that you have purchased for your own use. It is NOT OK to make copies to sell or give to others. This is unfair to people who invest their time and money to provide good, inexpensive software, and is illegal as well.

CLOAD: FROM TAPE TO MEMORY



This chapter presents a new way to use the recorder. You will learn how to save data (information) on cassettes, and how to read data from cassettes into the memory of the TRS-80.

The data (information you store on tape cassettes) can consist of numbers or strings or a mixture of both. So, the information can be almost anything. For example:

- A personal telephone directory with people's names and phone numbers.
- A dictionary of three-letter words to be used in a computer game.
- An inventory of your record, coin, or stamp collection — or whatever you collect.
- A list of your important personal property. Put this cassette in your bank deposit box. You might need it if your house burns down or you are burgled!
- The first five hundred prime numbers.
- Your shopping list for next Christmas (add to it now and then).
- People's birthdays, anniversaries, and other important dates.
- Tax information so you and your friendly TRS-80 can go bravely into battle against the giant IRS monster.

Why put such information on tape cassettes? Because, once it is on cassettes you can read it into your TRS-80 and do things with it, or to it, as the case may be. In addition, information stored on cassette tape is "machine-readable." The TRS-80 automatically reads it much faster (and with fewer errors) than you can type it in. So, save wear and tear on the old fingers — learn how to put data on cassette tape.

IT'S EASY!

Start with two short, simple programs. The first program lets you enter information from the keyboard and save it on tape. Of course, this information must first go into the memory (RAM) of the TRS-80. That's why we call this program KEYBOARD TO MEMORY TO TAPE.

```
10Ø REM ** KEYBOARD TO MEMORY TO TAPE
11Ø CLS
12Ø INPUT A$ <-------------- Keyboard to memory
13Ø PRINT # -1, A$ <-------- Memory to tape
14Ø GOTO 12Ø
```

Line 120 is the KEYBOARD TO MEMORY part of the program. When a question mark appears, you enter a string. The string is stored into memory as the value of A$.

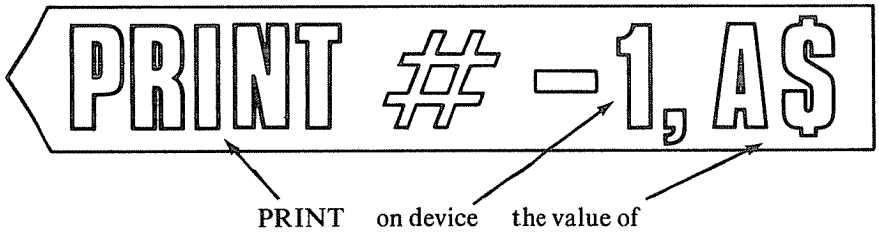
Line 130 is the MEMORY TO TAPE part of the program.

The statement:

```
13Ø PRINT # -1, A$
```

tells the TRS-80 to "PRINT" the value of A$ on device # −1, which just happens to be the cassette tape recorder.

Let's say it again, boldly.



Now you have two ways to use PRINT.

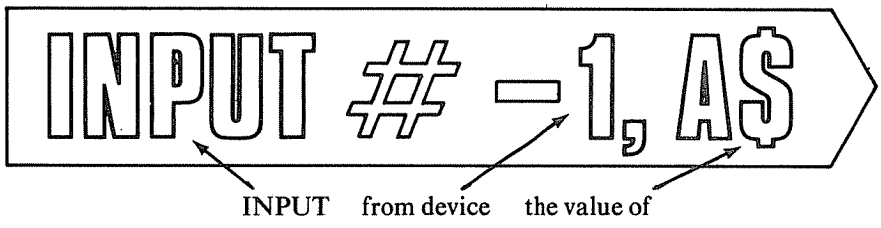
1) PRINT A$ means to "PRINT" on the TV screen.
2) PRINT #−1, A$ means to "PRINT" on the cassette tape.

As you work through this book, you will find even more ways to use PRINT.

OK, you have a program to put information on tape. Here is a program to get information from tape into memory. This program is called TAPE TO MEMORY TO SCREEN.

```
2ØØ REM ** TAPE TO MEMORY TO SCREEN
21Ø CLS
22Ø INPUT # -1, A$ <------- Tape to memory
23Ø PRINT A$ <------------- Memory to screen
24Ø GOTO 22Ø
```

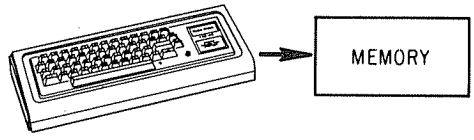
Line 220 is the TAPE TO MEMORY part of the program. It tells the TRS-80 to INPUT a value from device # −1 (AHA! The cassette recorder) and put it into A$.



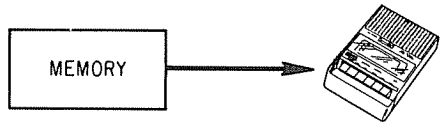
Line 230 is the MEMORY TO SCREEN part of the program. It tells the TRS-80 to PRINT the value of A$ (which is in memory) on the TV screen.

In summary:

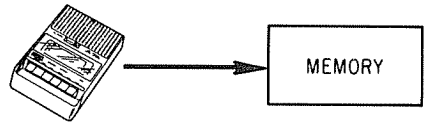
INPUT A$



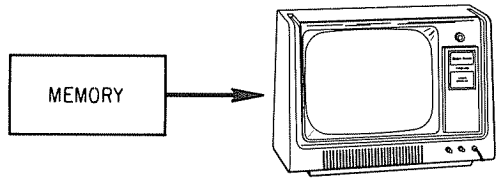
PRINT # −1, A$



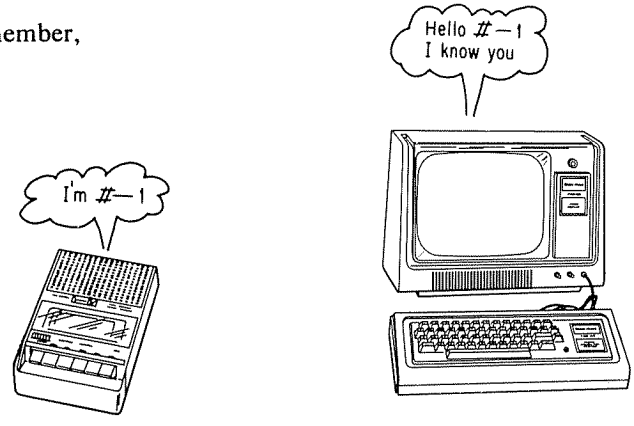
INPUT # −1, A$



PRINT A$



And remember,



SLOW DOWN AND . . . READ VEERRRYYYY CAREFULLY!

Now we are going to try out our two programs. This must be done VERY carefully. We wish you success on your very first try. Read slowly, then reread, then read again. S l o w is good.

Start by finding a high quality, never-before-used tape cassette. Don't (repeat, DON'T) buy just any old cheap cassette. Always treat your TRS-80 to the best, if you want the best from it. Next, examine the cassette. Most cassettes have lots of magnetic tape and . . . VERY IMPORTANT . . . a few inches of *leader.* Leader? Rewind your tape. The first few inches probably consists of *nonmagnetic* leader, usually clear, yellow, or blue, or any color other than dull brown. Dull brown is the color of magnetic tape.

YOU CAN'T RECORD ON LEADER

Why put all this space into what you can't do on leader? Because, if you want your TRS-80 to CSAVE or PRINT # −1, then you must

- rewind the tape,
- then, wind it forward a few inches so that magnetic tape, *not* leader, is in position to receive your message.

Now, enter both of our programs into the TRS-80. Note how we have cleverly chosen line numbers so that you can do this. For your convenience, here again are the two programs.

```
10Ø REM ** KEYBOARD TO MEMORY TO TAPE
11Ø CLS
12Ø INPUT A$
13Ø PRINT # -1, A$
14Ø GOTO 12Ø

2ØØ REM ** TAPE TO MEMORY TO SCREEN
21Ø CLS
22Ø INPUT # -1, A$
23Ø PRINT A$
24Ø GOTO 22Ø
```

We assume that you have placed the cassette in the cassette recorder. You have, haven't you? What! You haven't? Well, OK, do it now.

- Press *both* PLAY and RECORD on the cassette recorder. Nothing should happen. If the tape starts to move, you probably don't have the recorder properly hooked up to the TRS-80 (check the plug).
- Set the volume control on the cassette recorder to 5 or 6, or whatever works when you use CSAVE and CLOAD.
- Type RUN and press ENTER.
  This is what you see.
  The TRS-80 is waiting
  for a value of A$.

```
?
```

- Now, type a string value for A$ and press ENTER. You should see the tape move in the cassette recorder. Aha! The TRS-80 has accepted your value, stored it in A$, then recorded it on tape.

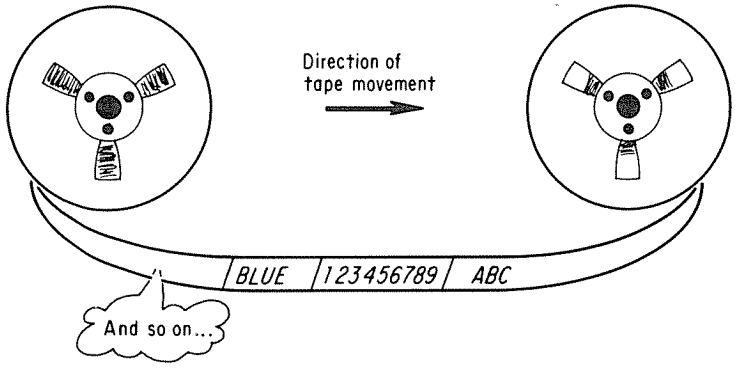
The following brief history shows what might happen when you run KEYBOARD TO MEMORY TO TAPE. Make this tape and keep it for later use.

```
? ABC
? 1234567890
? BLUE
? COMPUTERTOWN USA!
? TAKE A DRAGON TO LUNCH
? TESTING
? ONE
? TWO
? THREE
? THAT'S ALL FOLKS!
?
READY
```

Press BREAK to stop the computer.

After typing each string, keep one eye on the cassette recorder. Each time you press ENTER, the tape should move. The TRS-80 is recording your last entry on the cassette.

If all went well, you recorded 10 strings on the tape cassette like this:



To find out if those 10 strings are really on tape, use the following TAPE TO MEMORY TO SCREEN program:

- First, press STOP on the cassette recorder.
- Rewind the tape. It is *not* necessary to move the tape forward because you are going to read *from* the tape, not record *onto* tape. In reading *from* tape, the TRS-80 ignores the leader.
- Press BREAK to stop the program in progress.
- Type RUN 210 and press ENTER. This will RUN the program, *beginning at line 210.* Another way to do this is to type GOTO 210 and press ENTER.

Poof! The screen goes blank. Nothing seems to be happening.

- Press PLAY on the cassette recorder. The tape begins to move; information appears on the screen.

Here's what happens when you run the program. First, the screen looks like this:

```
ABC
```

Good! That's the first string you put on tape.

A little later, the screen looks like this:

```
ABC
1234567890
BLUE
COMPUTERTOWN USA!
TAKE A DRAGON TO LUNCH
```

IMPORTANT NOTICE! This is what happens if all goes well. You *did* use a high quality, never-before-used tape, didn't you? If not, you may be getting "noise" on the screen!

And still later:

```
ABC
1234567890
BLUE
COMPUTERTOWN USA!
TAKE A DRAGON TO LUNCH
TESTING
ONE
TWO
THREE
THAT'S ALL FOLKS!
```

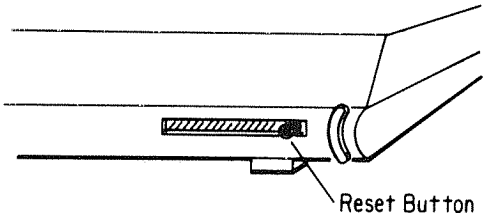
You may recall that that is everything you put on tape. But glance at the cassette recorder — the tape is still moving. You know that there is no more information on the tape. But the TRS-80 doesn't know. Look at the program.

```
200 REM ** TAPE TO MEMORY TO SCREEN
210 CLS
220 INPUT # -1, A$
230 PRINT A$
240 GOTO 220
```

It's still trying!

Interrupt your patient TRS-80 and you can try some new things.

- Press STOP on the cassette recorder. This stops the tape. Remove the cassette if you wish.
  The TRS-80 is *still* trying to read tape! Press the RESET button, located in the back of the keyboard unit, left side as you face the keyboard. It is behind a little plastic door.



Rear of Keyboard Unit

We hope everything went well. If it didn't, reread our directions carefully, then try again. Do each step s l o w l y and carefully. If you still have trouble, try these remedies.

- Clean and demagnetize the heads on your cassette recorder.
- Check the volume setting. A setting of 4 to 6 should work. Try several settings in this range.
- Check the cables that connect the cassette recorder to the keyboard unit.
- Try a fresh, good-quality cassette.
- Yell for HELP! With so many TRS-80s in use, there is probably another user within shouting distance.
- Take this chapter and your TRS-80 to the nearest Radio Shack store. They will be happy to help you.

On Stopping when Done

Here, once again, are the two programs KEYBOARD TO MEMORY TO TAPE and TAPE TO MEMORY TO SCREEN. Well, almost — we have made a small change in the second program.

```
10Ø REM ** KEYBOARD TO MEMORY TO TAPE
11Ø CLS
12Ø INPUT A$
13Ø PRINT # -1, A$
14Ø GOTO 12Ø

2ØØ REM *** TAPE TO MEMORY TO SCREEN
21Ø CLS
22ØINPUT # -1, A$
23Ø PRINT A$

235 IF A$ = "THAT'S ALL FOLKS!" THEN END

24Ø GOTO 22Ø
```

Type RUN 210 or GOTO 210 to use this one.

Enter this program, then rewind your tape from page 70, type RUN 210, and press ENTER.

```
ABC
1234567890
BLUE
COMPUTERTOWN USA!
TAKE A DRAGON TO LUNCH
TESTING
ONE
TWO
THREE
THAT'S ALL FOLKS!
READY
```

Success! The program stopped, automatically. The tape stopped, too. Just as you expected. . . ?

The string THAT'S ALL FOLKS! marks the *end of file* on the tape. It marks the end of information on the tape. Well, actually, it marks the end of this particular bunch of information on tape. If you want to put more information on this tape, you may. But *you* must remember where it begins and where it ends. Use the tape counter on the cassette recorder to position the tape at the beginning of your information.

- A bunch of information on tape is sometimes called a *record.*
- A bunch of records is called a *file.*

Oh well, whatever they are called, let's now set up something useful; a personal phone directory. Again, we have two programs; one to set up the directory and one to play it back. There is nothing new in these programs. Read them, figure out what they do, then move on.

By now, you should read the programs carefully and try to understand them *before* you try them on your TRS-80. Who knows? You might improve them before trying them!

```
100 REM ** MAKE A TAPE OF NAMES AND NUMBERS
110 CLEAR 100
120 CLS
130 PRINT : INPUT "NAME" ; NAYM$
140 INPUT "NUMBER" ; NMBR$

150 PRINT # -1, NAYM$, NMBR$

160 GOTO 130

200 REM ** READ 10 NAMES AND NUMBERS
210 CLEAR 100
220 CLS

230 FOR K = 1 TO 10
240   INPUT # -1, NAYM$, NMBR$
250   PRINT NAYM$ TAB(30) NMBR$
260   IF NAYM$ = "NO MORE  NAMES" THEN END
270 NEXT K

300 REM ** THE PRINT STATEMENTS TELL ALL
310 PRINT
320 PRINT "TO GET MORE NAMES AND NUMBERS,"
330 PRINT "PRESS THE SPACE BAR" ;
340 KEY$ = INKEY$ : IF KEY$ = "" THEN 340
350 IF KEY$ = " " THEN 220 ELSE 340
```

Sorry, NAME is a reserved word

Aha! Two things in one PRINT # −1

Put a space between quotes

QUESTION: When you run the program called MAKE A TAPE OF NAMES AND NUMBERS, what should you put on the tape as the *very last* "name?" What do you suggest for the corresponding telephone NMBR$? (We have two completely outrageous numbers in mind.) OK, let's go! *We* ran program MAKE A TAPE OF NAMES AND NUMBERS. If you had been there to watch, this is what you would have seen.

We typed RUN (or RUN 100 or RUN 110 or GOTO 100 or GOTO 110) and . . .

```
NAME? DON INMAN
NMBR? 415-323-6117

NAME? RAMON ZAMORA
NMBR? 415-323-6117

NAME? BOB ALBRECHT
NMBR? (415) 323-6117

NAME? COMPUTERTOWN USA!
NMBR? 415/323-3111

NAME? DYMAX GAZETTE
NMBR? 4153236117

NAME?
```

From now on, each new line pushes the line at the top off the screen. This does not affect the writing of information onto the cassette tape.

We continue.

```
NAME? GANDALF
NMBR? 777 777 7777

NAME? PRIME TIME
NMBR? 235 711 1317

NAME? FIBONACCI
NMBR? 112 358 1321

NAME? PCC
NMBR? 415 323 3111

NAME? ERIC BAKALINSKY
NMBR? 415 DAFFODIL

NAME? TIME OF DAY
NMBR? POPCORN

NAME? INFORMATION
NMBR? AREA CODE 555-1212

NAME? HANDICAPPED AID
NMBR? 800 722-3240

NAME? NO MORE NAMES
NMBR? 999 999 9999

NAME?

READY
>-
```

We are finished. So, we press STOP on the recorder and BREAK on the keyboard.

We arbitrarily chose 999 999 9999 as the number for NO MORE NAMES. Hmmm . . . could it be an actual number? We tried it. This is what we heard:

> The number you dialed is not a working number.
> Please check your listing and dial again.

Great! We had hoped that this would not be a real telephone number.

Now, let's play it back. We assume, of course that:

- the programs (both) are in memory and ready to go, and
- the cassette containing the file of names and phone numbers is in the cassette recorder and ready to go (rewound? right side up? OK!).

GO! Type RUN 210 and press ENTER. This is what you see:

```
DON INMAN             415-323-6117
RAMON ZAMORA          415-323-6117
BOB ALBRECHT          (415) 323-6117
COMPUTERTOWN USA!     415/323-3111
DYMAX                 4153236117
GANDALF               777 777 7777
PRIME TIME            235 711 1317
FIBONACCI             112 358 1321
PCC                   415 323 3111
ERIC BAKALINSKY       415 DAFFODIL

          TO GET MORE NAMES AND NUMBERS,
          PRESS THE SPACE BAR.
```

Now, press the space bar. The screen clears and the next bunch of names and numbers come on the screen.

```
TIME OF DAY           POPCORN
INFORMATION           AREA CODE 555-1212
HANDICAPPED AID       800 772-3240
NO MORE NAMES         999 999 9999

READY
>_
```

Since there are NO MORE NAMES on the cassette, the TRS-80 stops automatically.

We put names on the cassette in no particular order. You might want to set up your personal telephone directory in alphabetical order as names usually appear in a printed directory. This might be especially useful if your tape directory has many names and numbers.

You Can Store Numbers, Too

So far, you have stored only string values on tape cassettes by using string variables in PRINT # −1 statements. You have used the following PRINT statements:

```
130 PRINT # -1, A$          <-- String variable
150 PRINT # -1, NAYM$, NMBR$
                            <-- String variables
```

We can also store numbers on tape cassettes by using numerical variables or expressions in the PRINT # −1, statement.

```
140 PRINT # -1, X   <-- Numeric variable
```

The following program will store consecutive integers (1, 2, 3, 4, 5, and so on) on a tape cassette.

```
100 REM ** MEMORY TO SCREEN AND TAPE
110 CLS
120 X = 1

130 PRINT X ;          <-- Memory to screen

140 PRINT # -1, X      <-- Memory to tape

150 X = X + 1
160 GOTO 130
```

This program also PRINTs each number on the screen (line 130) just before writing it on tape (line 140).

And, of course, you also need a program to read the numbers back from tape into the memory and put them on the screen.

```
200 REM ** TAPE TO MEMORY TO SCREEN
210 CLS
220 INPUT # -1, X
230 PRINT X ;
240 GOTO 220
```

Enter both programs. Then, put a C-10 cassette (five minutes on each side) in the recorder, press PLAY and RECORD, type RUN on the keyboard, press ENTER . . . and this is what you see:

```
 1  2  3  4  5  and so on
```

This goes on for about five minutes until the end of the tape is reached. CLICK! The recorder stops, but the TRS-80 continues putting numbers on the screen.

We now press the reset button to stop the TRS-80.

Let's find out how many numbers are on the tape. Rewind the tape, press PLAY on the recorder, type RUN 210 and press ENTER. When the recorder stops, press the reset button.

The numbers are read one by one as the value of X appears on the screen. Five minutes pass . . . CLICK! The recorder stops at the end of the tape.

```
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18
 19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
 35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
 51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66
 67  68  69  70  71  72  73  74  75

READY
>-
```

The recorder stopped here. The TRS-80 is still trying to read tape. Press the reset button to stop it.

Using the statement PRINT # −1, X allows us to put about seventy-five numbers on one side of a C-10 cassette. You might get a slightly different number on your C-10 cassette. However, assuming about seventy-five numbers on a C-10, we might expect about the following on long cassettes.

CASSETTE	NUMBER OF NUMBERS
C-30	225
C-60	450
C-90	675

A C-30 has fifteen minutes on each side, a C-60 has thirty minutes on each side, and so on.

Writing only one number at a time is the least efficient way to store information on a cassette. How can we store more numbers on tape? Try this: Make these changes to the programs MEMORY TO SCREEN AND TAPE and TAPE TO MEMORY TO SCREEN.

```
140 PRINT # -1, X, X + 1
150 X = X + 2
220 INPUT # -1, X, Y
230 PRINT X; Y;
```

- Use a fresh C-10 tape and RUN the (modified) program called MEMORY TO SCREEN AND TAPE.
- Then, play this tape back, using the program called TAPE TO MEMORY TO SCREEN. Remember, to read this tape back, rewind it, then type RUN 210 or GOTO 210 to start the program.

Using a C-10 tape, we ran both programs. Here is what we got when we played the tape back, using the second program:

```
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18
 19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
 35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
 51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66
 67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82
 83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98
 99  100  101  102  103  104  105  106  107  108  109  110  111
 112  113  114  115  116  117  118  119  120  121  122  123
 124  125  126  127  128  129  130  131  132  133  134  135
 136  137  138  139  140  141  142  143  144  145  146  147
 148
READY
```

The recorder stopped. We pressed the reset button to stop the TRS-80.

Aha! This time, the C-10 tape has 148 numbers, almost twice as many as before. This happened because *two* numbers are written on tape each time.

```
140 PRINT # -1, X, X + 1
```

two numbers

If we write two numbers on tape each time, we usually INPUT two numbers when we play back the tape.

```
220 INPUT # -1, X, Y
```

two numbers (put one in X and the other in Y).

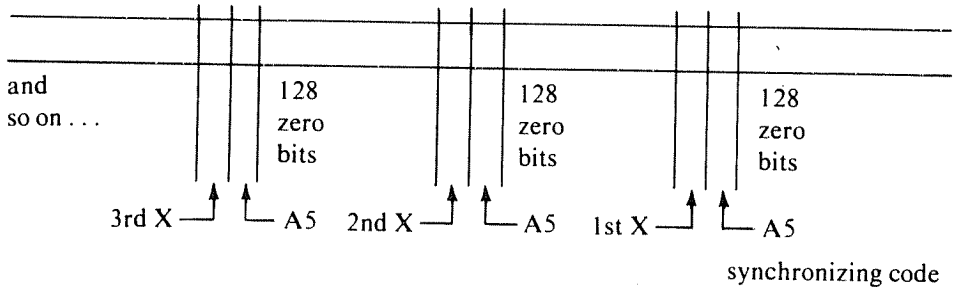
Hmmm . . . what happens if we INPUT only one number? Try it and find out!

Technical Stuff

You may skip this section if you wish. We will explain how the TRS-80 writes information on a cassette tape; then you may understand why PRINT # −1, X, X+1 crams about twice as much information on tape as PRINT # −1, X.

In executing a PRINT # −1 statement, the TRS-80 first records 128 zero bits (binary digits) on the tape. It then records a code which is later used to sychronize the tape and the computer when the tape is read back. This code is 1010 0101 in binary or A5 in hexadecimal. Huh? Well, we did say you could skip this section!

As you might expect, all those zero bits and the hexadecimal code use up some tape. Finally, however, *your* information gets recorded on tape. So, if you are putting only one number on tape at a time, it looks something like this:

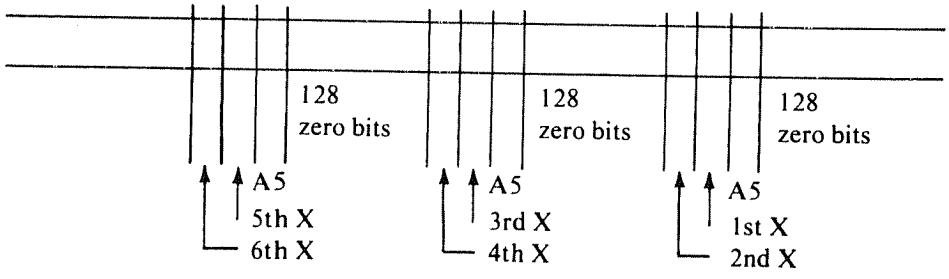


When you save the pieces of data in the same PRINT statement, such as:

```
140 PRINT # -1, X, X+1
```

two numbers

approximately twice as much data can be stored on the same length of tape.



By comparing this sketch and the previous sketch, you can see that more data can be squeezed onto the same length of tape by printing more than one item in a PRINT # −1 statement. However, there is a limit to the number of items given in the PRINT # − statement. The total number of characters printed by one PRINT # −1 statement *must not* exceed 255. All characters after the 255th would be cut off or ignored.

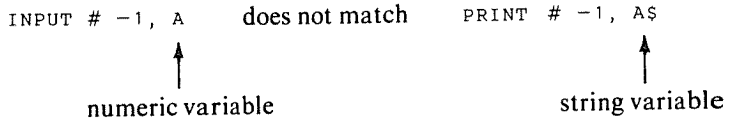
The characters would be counted like this:

1st X = 2	1 character	
2nd X = 32	2 characters	
3rd X = 55.9	4 characters	(the decimal point counts as a character)
TOTAL	7 characters	

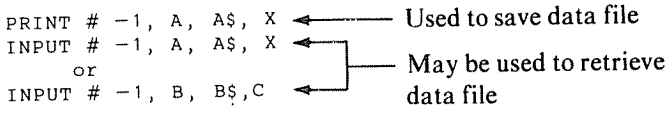
When using PRINT # −1 and INPUT # −1 statements, you must be sure that the variables used in both statements match. If numeric variables are saved, then numeric variables must be used to retrieve that data. If string variables are saved, then string variables must be used to retrieve that data.

`INPUT # -1, A`	matches	`PRINT # -1, A`	(the same variable)
`INPUT # -1, A`	matches	`PRINT # -1, X`	(same *type* of variable)
`INPUT # -1, A$`	matches	`PRINT # -1, A$`	(same variable)
`INPUT # -1, A$`	matches	`PRINT # -1, B$`	(same *type* of variable)

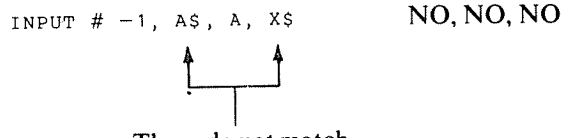
However,



If you mix variable types in `PRINT # -1` statements, use great care to match the variable types in the `INPUT # -1` statement.



But not



Well that's enough about cassette data files for one chapter. Stop now and read through the summary. Then try the Self-Test before going on.

Summary

Even though you had probably used the cassette recorder before to CSAVE and CLOAD programs, this chapter introduced some new uses. You learned:

- to transfer data from memory to tape.

```
PRINT # -1 , A$
```

PRINT — on device — the string

or

```
PRINT # -1 , X
```

the numeric value of X

- that more than one data item can be saved on tape by the same PRINT # −1 statement.

```
PRINT # -1, A, B, C
```

three numeric values

- to transfer data from tape to memory.

```
INPUT # -1 , A$
```

INPUT — from device — the string A$

or

```
INPUT # -1 , X
```

The numeric value to be assigned to X

- if data has been saved on tape using more than one data item per PRINT # −1 statement, it *must be* input in the same format.

If you saved it this way:

```
PRINT # -1, A,B,C
```

three numeric values

it must be input this way,

```
INPUT # -1, A,B,C
```

three numeric variables.

(The variable names do not have to be the same ones used in PRINT # −1, but they must be the same type., i.e., X,Y,Z would be OK, but NOT X$, Y$, Z$)

- that the amount of tape used to save data files can be conserved by printing more than one item in a PRINT # −1 statement.
- that the total number of characters saved by one PRINT # −1 statement must not exceed 255.

Self-Test

1. When we CLOAD a program, the program is read from a cassette into memory. Is the program read into ROM or into RAM? _____
2. When you CSAVE a program, the TRS-80 records the program in its memory onto a cassette. Is the program copied from ROM or from RAM? _____
3. Is it possible to enter information from a cassette into ROM? _____
4. Is it possible to copy information from ROM onto a cassette? (Go ahead-guess!) _____

   If you said *yes*, please explain how this might be done. _____
5. Suppose we RUN the following program.

```
10Ø REM ** MYSTERY PROGRAM 2A-1
11Ø A$ = "BLUE" : B$ = "YELLOW" : C$ = "RED"
12Ø CLS
13Ø PRINT "PRESS PLAY AND RECORD ON CASSETTE # -1"
14Ø PRINT "THEN PRESS THE SPACE BAR"
15Ø KEY$ = INKEY$ : IF KEY$ = "" THEN 15Ø
16Ø IF KEY$ = " " THEN 17Ø ELSE 15Ø
17Ø PRINT # -1, A$, B$, C$
999 END
```

   (a) Show what first appears on the screen.

(b) If you press the space bar, what happens? _____

_____

_____

6. Here are our first KEYBOARD TO MEMORY TO TAPE and TAPE TO MEMORY TO SCREEN programs. Rewrite them so that instructions to the operator are given, as in exercise 5, above.

```
100 REM ** KEYBOARD TO MEMORY TO TAPE
110 CLS
120 INPUT A$
130 PRINT # -1, A$
140 GOTO 120

200 REM ** TAPE TO MEMORY TO SCREEN
210 CLS
220 INPUT # -1, A$
230 PRINT A$
240 GOTO 220
```

7. Explain the following program:

```
10Ø REM ** MEMORY TO TAPE
11Ø CLS
12Ø READ A$
13Ø PRINT # -1, A$
14Ø IF A$ = "THAT'S ALL FOLKS!" THEN END
15Ø GOTO 12Ø
16Ø DATA ABC, 1234567890, BLUE, COMPUTERTOWN USA!
17Ø DATA TAKE A DRAGON TO LUNCH, TESTING, ONE, TWO, THRE E
18Ø DATA "THAT'S ALL FOLKS!"
```

8. Write a program to put the first twenty-five prime numbers on a cassette as *numeric* (not string) values. The first twenty-five prime numbers are shown below.

```
 2   3   5   7  11  13  17  19  23  29
31  37  41  43  47  53  59  61  67  71
73  79  83  89  97
```

9. Write a program to put on tape the names and ages of the people below. Use a string variable for the name and a numeric variable for the age. Make five records, each containing one name and one age.

JONES, 45
KEYS, 22
LARS, 27
MUNZ, 52
NANCE, 18

10. Write a program to transfer from tape to memory to screen the data saved in exercise 9.

Answers to Self-Test

1. RAM
2. RAM
3. No
4. Yes. You could assign a variable to a PEEK statement such as A = PEEK(25), then PRINT # −1, A.
5. (a)
```
PRESS PLAY AND RECORD ON CASSETTE # -1
THEN PRESS THE SPACE BAR
```
   (b) The tape moves and the data for A$, B$, and C$ is recorded. The tape then stops.

6. This is one way to do it. Yours may be different. If yours works, it is correct. Try it.

```
100 REM ** KEYBOARD TO MEMORY TO TAPE
110 CLS
120 PRINT "INPUT A STRING WHEN THE ? APPEARS."
130 PRINT "THEN PRESS THE ENTER KEY."
140 PRINT "IF ALL ENTRIES HAVE BEEN MADE TYPE 'DONE' "
150 PRINT "FOR YOUR INPUT."
160 INPUT A$
170 PRINT # -1, A$
180 IF A$ = "DONE" THEN END
190 GOTO 160

200 REM ** TAPE TO MEMORY TO SCREEN
210 CLS
220 PRINT "PRESS PLAY AND RECORD ON CASSETTE # -1."
220 PRINT "THEN PRESS THE SPACE BAR."
230 KEY$ = INKEY$, IF KEY$ = "" THEN 230
240 IF KEY$ = " " THEN 250 ELSE 230
250 INPUT # -1, A$
260 PRINT A$
270 IF A$ = "DONE" THEN END
280 GOTO 250
```

7. The program reads the items in the data list one at a time, turns on the recorder, and saves the item read. The recorder then stops. This is repeated for each data item. When the last item is recorded the program stops.

8. Here is one program. Yours, of course, may be different. Try yours to make sure it works.

```
100 CLS
110 FOR N = 1 TO 100
120   M = 2
130   IF N/M<>INT(N/M) THEN M = M + 1 : GOTO 130
140   IF N = M THEN PRINT # -1, N
150 NEXT N
```

9.
```
100 CLS
110 FOR N = 1 TO 5
120   INPUT A$, A
130   PRINT # -1, A$, A
140 NEXT N
```

10.
```
200 CLS
210 FOR N = 1 TO 5
220   INPUT # -1, A$, A
230   PRINT A$; "," ; A
240 NEXT N
```

CHAPTER FIVE

More About Cassette Files

In this chapter, you will learn more ways to copy information from the memory of the TRS-80 onto cassette tapes and read information from these tapes into the TRS-80's memory. You will learn to:

- put information on a cassette tape by using the READ, DATA, and PRINT #1−1 statements,
- put information stored in a numeric or string array onto cassette tape, and
- read information from a cassette tape into an array.

Something Old, Something New

In chapter 4, you started with two programs called KEYBOARD TO MEMORY TO TAPE and TAPE TO MEMORY TO SCREEN. Here they are again:

```
10Ø REM **KEYBOARD TO MEMORY TO TAPE
11Ø CLS
12Ø INPUT A$
13Ø PRINT #-1, A$
14Ø GOTO 12Ø

2ØØ REM ** TAPE TO MEMORY TO SCREEN
21Ø CLS
22Ø INPUT #-1, A$
23Ø PRINT A$
24Ø GOTO 22Ø
```

You used the first program (lines 100 through 140) to put information on tape (see page 67). Here's a different program to put information on tape.

```
100 REM ** READ FROM MEMORY, PUT ON TAPE
110 CLS
120 READ A$
130 PRINT #-1, A$
140 GOTO 120
```

Aha! Instead of getting the value of A$ from the keyboard, the TRS-80 will get it from a DATA statement. So, let's put the required information in DATA statements, as shown in the following program:

```
1000 REM ** INFORMATION TO PUT ON TAPE
1010 DATA ABC
1020 DATA 1234567890
1030 DATA BLUE
1040 DATA COMPUTERTOWN USA!
1050 DATA TAKE A DRAGON TO LUNCH
1060 DATA TESTING
1070 DATA ONE
1080 DATA TWO
1090 DATA THREE
1100 DATA THAT'S ALL FOLKS!
```

We put one item of information in each DATA statement. Of course, you can pack more information per DATA statement if you wish. For example, it could be like this:

```
1000 REM ** INFORMATION TO PUT ON TAPE
1010 DATA ABC, 1234567890, BLUE
1020 DATA COMPUTERTOWN USA! TAKE A DRAGON TO LUNCH
1030 DATA TESTING, ONE, TWO, THREE
1040 DATA THAT'S ALL FOLKS!
```

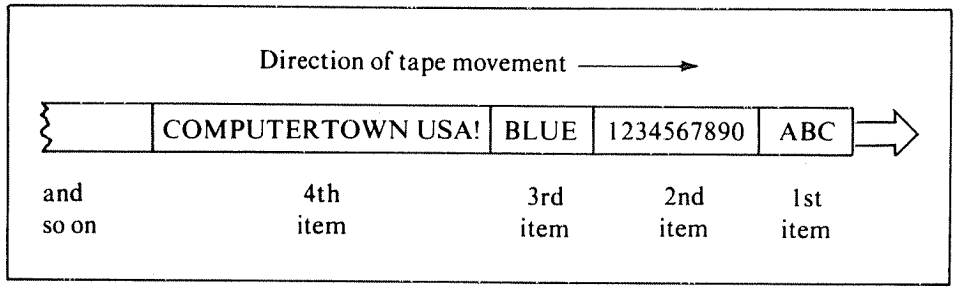
We arbitrarily began our block of information at line 1000. You can choose a different block of line numbers.

By now, you can put a fresh, blank tape in the recorder and get everything ready to go. Do it! Then, run the READ FROM MEMORY, PUT ON TAPE program. CLICK, WHRRR; CLICK, WHRRR; . . . and so on, as information is READ from DATA statements one item at a time and put on tape.

Eventually, the last item in the last DATA statement is read and put on tape. The TRS-80 stops with an OUT OF DATA message on the screen.

```
?OD ERROR IN 120
READY
>-
```

The information in the DATA statements has been recorded on a cassette tape. If all went well, 10 strings are on the tape, as follows:



Did it happen? Find out by playing the tape back, using the program TAPE TO MEMORY TO SCREEN, just as you did in the preceding chapter (pages 78-80).

Hmmm . . . try the tape again using this version of TAPE TO MEMORY TO SCREEN.

```
200 REM ** TAPE TO MEMORY TO SCREEN
210 CLS
220 INPUT #-1, A$
230 PRINT A$
235 IF A$ = "THAT'S ALL FOLKS!" THEN END  <-- Add this statement
240 GOTO 220
```

Numbers Instead of Strings

Instead of string, let's put numbers on tape. The following version of READ FROM MEMORY, PUT ON TAPE uses a numeric variable (X) in lines 120 and 130.

```
100 REM ** READ FROM MEMORY, PUT ON TAPE
110 CLS
120 READ X  <-- X is a numeric variable
130 PRINT #-1, X
140 GOTO 120

1000 REM ** NUMBERS TO PUT ON TAPE
1010 DATA 1, 2, 3, 4, 5
1020 DATA 6, 7, 8, 9, 10
1030 DATA 11, 12, 13, 14, 15
1040 DATA 16, 17, 18, 19, 20
```

Are the numbers on tape? Find out. Play the tape back, using this program which you may recall from chapter 4.

```
200 REM ** TAPE TO MEMORY TO SCREEN
210 CLS
220 INPUT #-1, X
230 PRINT X;
240 GOTO 220
```

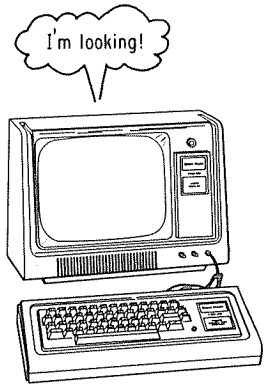
If everything has been done right, the numbers from 1 to 20 are read from the tape and the screen looks like this.

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20
```

At this point, the tape keeps running but no more numbers are printed on the screen.

Of course! There *aren't* any more numbers on the tape, but your ever-patient TRS-80 keeps looking for more numbers because that's what you told it to do.

```
210 CLS
220 INPUT #-1, X
230 PRINT X;
240 GOTO 220
```



It isn't going to find more numbers, so:

- Press STOP on the tape recorder.
- Press the reset button on the TRS-80.

Flags to the rescue! Change the last DATA statement, as follows.

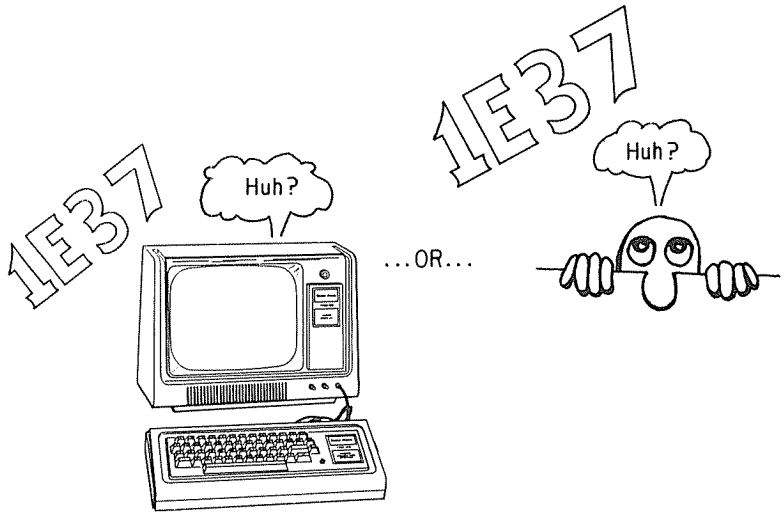
```
1040 DATA 16, 17, 18, 19, 20, -1
```

An end of data flag.

Then, add this line to the TAPE TO MEMORY TO SCREEN program.

```
225 IF X = -1 THEN END
```

Try the above changes. The flag (−1) is the last thing on tape. It is detected when you read the tape back into memory and causes the program to stop. If you want the flag (−1) to be printed on the screen, use 235 as the line number of the IF statement that looks for the flag. Feel free to choose a different number as the flag. A good choice might be 1E37, a very unlikely number!



Nayms and Numbers

Do you remember the personal telephone directory program in the preceding chapter? If not, go back and browse through it again. You can now make an even more useful telephone directory by using arrays to hold the names and phone numbers. First, enter the names and numbers into two arrays, NAYM$ (for the names) and NMBR$ (for the telephone numbers). Use the CLEAR statement to reserve enough memory space to hold your names and numbers.

The following program segment allows you to enter names and numbers from the keyboard and store them in the array NAYM$ and NMBR$.

```
100 REM ** PERSONAL TELEPHONE DIRECTORY
110 REM ** SET UP STRING ARRAYS NAYM$ AND NMBR$
120 REM ** NAYM$ = NAMES NMBR$ = PHONE NUMBERS
130 CLEAR 5000
140 DIM NAYM$(200), NMBR$(200)
```

Arrays NAYM$ and NMBR$ can store up to two hundred names and numbers with a maximum total of five thousand characters. If you need more, change lines 130 and 140.

```
200 REM**KEYBOARD TO NAYM$ AND NMBR$
210 CLS
220 K = 1
230 PRINT : INPUT "NAME " ; NAYM$(K)
240 INPUT "NUMBER" ; NMBR$(K)
250 IF NAYM$(K)="ZZZZZ" THEN 310
260 K = K + 1 : GOTO 230
```

RUN the program and enter these names and numbers in alphabetical order. Do NOT include commas in strings. We will explain why later.

NAME	NUMBER	
Albrecht Bob	415 232 6117	
COMPUTERTOWN USA!	415 323 3111	
Inman Don	415 323 6117	
People's Computer Co.	415 323 3111	
Radio Shack	817 390 3011	
Zamora Ramon	415 327 0541	
ZZZZZ	999 999 9999	This is the "end-of-directory" flag.

It should go like this:

```
NAME? ALBRECHT BOB
NUMBER? 415 323 6117

NAME? COMPUTERTOWN USA!
NUMBER? 415 323 3111

NAME? INMAN DON
NUMBER? 415 323 6117

NAME? PEOPLE'S COMPUTER COMPANY
NUMBER? 415 323 3111

NAME? RADIO SHACK
NUMBER? 817 390 3011

NAME?
```

From now on, each new line will push a line off the top of the screen. This does not affect storing the information into the arrays NAYM$ and NMBR$. We continue:

```
.
.
.
NAME? ZAMORA RAMON
NUMBER? 415 327 Ø541

NAME? ZZZZZ
NUMBER? 999 999 9999      <-- End of directory

?UL ERROR IN LINE 25Ø
READY
>-
```

Oops! Check line 250 in the program. Aha! Since NAYM$(K) is equal to "ZZZZZ," the TRS-80 tried to go to line 310. Since there is, as yet, no line 310, it stopped with an Undefined Label error. We will fix that soon.

Our little phone directory is now stored in arrays NAYM$ and NMBR$, as follows.

NAYM$(1)	ALBRECHT BOB	NMBR$(1)	415 323 6117
NAYM$(2)	COMPUTERTOWN USA!	NMBR$(2)	415 323 3111
NAYM$(3)	INMAN DON	NMBR$(3)	415 323 6117
NAYM$(4)	PEOPLE'S COMPUTER CO	NMBR$(4)	415 323 3111
NAYM$(5)	RADIO SHACK	NMBR$(5)	817 390 3011
NAYM$(6)	ZAMORA RAMON	NMBR$(6)	415 327 0541
NAYM$(7)	ZZZZZ	NMBR$(7)	999 999 9999

You can use direct statement to convince yourself that the directory is stored. Clear the screen, then:

```
You type:     PRINT NAYM$(2)
It prints:    COMPUTERTOWN USA!

You type:     PRINT NMBR$(2)
It prints:    415 323 3111
```

Or, you can do it this way:

```
You type:     PRINT NAYM$(3), NMBR$(3)
It prints:    INMAN DON   415 323 6117
```

The next program segment stores the names and numbers on a cassette. First, it prompts you to press the play and record keys on the recorder. Of course, you have already put a fresh new cassette in the recorder, haven't you? If not, do it now.

```
300 REM ** INSTRUCTIONS TO USER
310 CLS
320 PRINT "PRESS PLAY AND RECORD ON THE RECORDER."
330 PRINT
340 PRINT "PRESS THE SPACE BAR AND I WILL STORE"
350 PRINT "THE NAMES AND PHONE NUMBERS ON TAPE."
360 K$ = INKEY$ : IF K$ = "" THEN 360
370 IF K$ = " " THEN 410 ELSE 360

400 REM ** NAYM$ AND NMBR$ TO TAPE
410 K = 1
420 PRINT # -1, NAYM$(K), NMBR$(K)
430 IF NAYM$(K)="ZZZZZ" THEN 450
440 K = K + 1 : GOTO 420
450 PRINT : PRINT "DONE. REWIND AND REMOVE TAPE."
460 END
```

Store and RUN the entire program, lines 100 through 450. The directory is now stored on tape; it is also still in memory.

We suggest you make a second copy on a second cassette. Remove the tape containing the first copy of the directory and position the second cassette in the recorder.

DON'T type RUN
DON'T type RUN 300

When you type RUN or RUN 300, the TRS-80 first *erases the values* of all variables. Bye, bye directory!

DO type GOTO 300

The TRS-80 will go to line 300 and begin. Sit back for a few moments as the computer puts a second copy of the directory on tape. Make as many copies as you wish.

Read and Use the Directory

Next, we need a program to read the personal telephone directory and to look up numbers. It begins like this:

```
1ØØ REM ** READ AND USE DIRECTORY
11Ø REM ** SET UP STRING ARRAYS NAYM$ AND NMBR$
12Ø REM ** NAYM$ = NAMES NMBR$ = PHONE NUMBERS
13Ø CLEAR 5ØØØ
14Ø DIM NAYM$(2ØØ), NMBR$(2ØØ)
```

Well, that certainly looks familiar, doesn't it? The next piece of the program reads the directory information from the tape cassette into NAYM$ and NMBR$.

```
2ØØ REM ** INSTRUCTIONS TO USER
21Ø CLS
22Ø PRINT "PRESS PLAY ON THE TAPE RECORDER."
23Ø PRINT
24Ø PRINT "PRESS ANY KEY AND I WILL READ THE TAPE"
25Ø PRINT "INFORMATION INTO MEMORY."
26Ø K$ = INKEY$ : IF K$ = "" THEN 26Ø

3ØØ REM **TAPE TO NAYM$ AND NMBR$
31Ø K = 1
32Ø INPUT #-1, NAYM$(K), NMBR$(K)
33Ø IF NAYM$(K)="ZZZZZ" THEN 35Ø
34Ø K = K + 1 : GOTO 32Ø
35Ø PRINT : PRINT "DONE. REWIND AND REMOVE TAPE."
36Ø PRINT : PRINT "TO CONTINUE, PRESS ANY KEY."
37Ø K$ = INKEY$ : IF K$ = "" THEN 37Ø
```

After you RUN this part of the program, the screen will look like this.

```
PRESS PLAY ON THE TAPE RECORDER.

PRESS ANY KEY AND I WILL READ THE TAPE
INFORMATION INTO MEMORY.

DONE. REWIND AND REMOVE TAPE.

TO CONTINUE, PRESS ANY KEY.
```

The directory is stored. Press any key and the TRS-80 will continue with the following section of the program:

```
4ØØ REM ** ASK FOR NAME TO LOOK-UP
41Ø CLS
42Ø INPUT "NAME" ; WHO$

5ØØ REM ** SEARCH NAYM$ ARRAY FOR WHO$
51Ø K = 1
52Ø IF NAYM$(K) = WHO$ THEN 61Ø
53Ø IF NAYM$(K) = "ZZZZZ" THEN 71Ø
54Ø K = K + 1 : GOTO 52Ø
```

If the name is in the directory, the TRS-80 will find it and go to line 610. If the name is not in the directory, the TRS-80 will search through the entire array NAYM$ and find the flag ZZZZZ. In this case, it will then go to line 710. Blocks 600 and 700 are shown below:

```
6ØØ REM ** PRINT THE NAME AND PHONE NUMBER
61Ø PRINT NAYM$(K), NMBR$(K)
62Ø PRINT
63Ø GOTO 42Ø

7ØØ REM ** OOPS! NAME NOT IN DIRECTORY
71Ø PRINT "THAT NAME IS NOT IN THE DIRECTORY."
72Ø PRINT
73Ø GOTO 42Ø
```

That's it. Enter the program and RUN it. The part of the program beginning at line 400 might go like this.

```
NAME? INMAN DON
INMAN DON  415 323 6117

NAME? ZAMORA
THAT NAME IS NOT IN THE DIRECTORY.

NAME? ZAMORA RAMON
ZAMORA RAMON  415 327 Ø541

NAME? and so on. . .
```

Hmmm . . . perhaps you are annoyed at having to type in the entire name *exactly as stored in the memory*. So, change block 500, as follows.

```
5ØØ REM ** SEARCH NAYM$ ARRAY FOR WHO$
51Ø LWHO = LEN(WHO$)
52Ø K = 1
53Ø IF LEFT$(NAYM$(K), LWHO) = WHO$ THEN 61Ø
54Ø IF NAYM$(K) = "ZZZZZ" THEN 71Ø
55Ø K = K + 1 : GOTO 53Ø
```

With this change, a RUN might look like this:

```
NAME? INMAN
INMAN DON  415 323 6117

NAME? RADIO
RADIO SHACK  817 39Ø 3Ø11

NAME? C
COMPUTERTOWN USA!  415 323 3111

NAME? Z
ZAMORA RAMON  415 327 Ø541

NAME? ZZ
ZZZZZ  999 999 9999

NAME? B
THAT NAME IS NOT IN THE DIRECTORY

NAME? and so on . . .
```

Now you can enter the left part of a stored name. The computer will find the first name for which the leftmost part of the name exactly matches your entry. If no match is found, the computer will eventually find the flag "ZZZZZ" and tell you that your name is not in the directory.

* Look at block 500. The statement 510 LWHO = LEN(WHO$) computes the length of the string WHO$ and assigns this value to LWHO. If the value entered for WHO$ is INMAN, then LWHO will be equal to 5.
* Then, look at line 530. The statement 530 IF LEFT$(NAYM$(K), LWHO) = WHO$ THEN 610 compares the leftmost LWHO characters of NAYM$(K) with WHO$. If they are equal, the TRS-80 goes to line 610. Otherwise, the TRS-80 goes to line 540.

No Commas, Please

Don't include commas in strings stored in cassette files, even if you enclose the strings in quotation marks. When you try to INPUT #−1, a string which includes a comma, you will get only the part of the string to the left of the comma.

Try this. First, enter the following program:

```
 1Ø CLS
 2Ø PRINT "PUT A FRESH TAPE IN THE RECORDER"
 3Ø PRINT "THEN PRESS PLAY AND RECORD"
 4Ø PRINT "NOW PRESS THE SPACE BAR"
 5Ø K$ = INKEY$ : IF INKEY$ = "" THEN 5Ø
 6Ø IF K$ = " " THEN 7Ø ELSE 5Ø

 7Ø PRINT #−1, "COMPUTERTOWN, USA!"
                                        <-- Aha! A comma
 8Ø PRINT
 9Ø PRINT "REWIND TAPE, THEN PRESS PLAY"
1ØØ PRINT "PRESS THE SPACE BAR"
11Ø K$ = INKEY$ : IF K$ = "" THEN 11Ø
12Ø IF K$ = "" THEN 13Ø ELSE 11Ø
13Ø INPUT−1, A$
14Ø PRINT A$
15Ø END
```

RUN the program. It should go like this:

```
PUT A FRESH TAPE IN THE RECORDER
THEN, PRESS PLAY AND RECORD
NOW, PRESS THE SPACE BAR

REWIND TAPE, THEN PRESS PLAY
PRESS THE SPACE BAR
?EXTRA IGNORED  <-- Older TRS-80s may not show this.
COMPUTERTOWN
READY.
>−
```

The TRS-80 stored "COMPUTERTOWN, USA!" on tape as *two* strings separated by a comma, *even* though the original string was enclosed in quotation marks.

Change lines 130 and 140 as follows:

```
13Ø INPUT #−1, A$, B$
14Ø PRINT A$ B$
```

With this change, the TRS-80 will read COMPUTER into A$ and USA! into B$. Try it.

Summary

You have learned in this chapter to:

- Put information on cassette tape by using READ, DATA, and PRINT #−1 statements.

```
 12Ø READ A$
 13Ø PRINT #−1, A$
 14Ø GOTO 12Ø
1ØØØ DATA ABC, 1234567890
```

- Put information stored in an array onto cassette tape.

```
41Ø K = 1
42Ø PRINT #−1, NAYM$(K) NMBR$(K)
43Ø K = K + 1 : GOTO 42Ø
```

- Read information from a cassette tape into an array.

```
31Ø K = 1
32Ø INPUT #−1, NAYM$(K), NMBR$(K)
33Ø K = K + 1 : GOTO 32Ø
```

- Store, retrieve, and use a Personalized Telephone Directory.

Self-Test

1. Names and telephone numbers are in DATA statements as follows.

```
1ØØØ REM ** INFORMATION TO PUT ON TAPE
1Ø1Ø DATA ALBRECHT BOB, 415 323 6117
1Ø2Ø DATA COMPUTERTOWN USA!, 415 323 3111
1Ø3Ø DATA INMAN DON, 415 323 6117
1Ø4Ø DATA PEOPLE'S COMPUTER COMPANY, 415 323 3111
1Ø5Ø DATA RADIO SHACK, 817 39Ø 3Ø11
1Ø6Ø DATA ZAMORA RAMON, 415 327 Ø541
1Ø7Ø DATA ZZZZZ, 999 999 9999
```

Write a program to READ the names and phone numbers in the DATA statements into arrays NAYM$ and NMBR$. Put the names into array NAYM$(1) through NAYM$(7) and the phone numbers into NMBR$(1) through NMBR$(7).

2. Using the DATA statements in exercise 1, write a program to store the names and phone numbers in the DATA statements directly onto a tape cassette. Do *not* put the information into arrays NAYM$ and NMBR$.

3. Our telephone directory program lets you look up a name, then gives you the phone number. Write a program segment, beginning at line 400, to ask for a number, then give the first name in the directory corresponding to that number.

4. Modify your program of exercise 3 so that after the first name is displayed,
   * pressing the space bar causes the computer to continue searching for the same phone number. If another occurrence is found, display it. If not, quit when ZZZZZ is found.
   * pressing the Q key (for Quit) causes the computer to return to the beginning of your program segment and ask for another number.

5. Modify the name look-up program blocks on pages 137 through 139 so that after name and phone number are found,
   * pressing the space bar causes the *next* name and number to be displayed.
   * pressing the Q key causes the computer to GOTO 410.

Answers to Self-Test

1.
```
 1ØØ REM ** PERSONAL PHONE DIRECTORY
 11Ø CLEAR 5ØØØ
 12Ø DIM NAYM$(2ØØ), NMBR$(2ØØ)

 2ØØ REM ** READ DATA & STORE IN ARRAYS
 21Ø CLS: K = 1
 22Ø READ NAYM$(K), NMBR$(K)
 23Ø IF NAYM$(K) = "ZZZZZ" THEN 25Ø
 24Ø K = K + 1 : GOTO 22Ø
 25Ø END
1ØØØ DATA (use data from problem)
```

or

```
 21Ø FOR K= 1 TO 7
 22Ø   READ NAYM$(K), NMBR$(K)
 23Ø NEXT K
 24Ø END
```

2.
```
 1ØØ REM ** STORE NAMES AND PHONE NUMBERS
 11Ø CLS : K = 1
 12Ø READ NAYM$, NMBR$
 13Ø PRINT #-1, NAYM$, NMBR$
 14Ø IF NAYM$ = "ZZZZZ" THEN 16Ø
 15Ø K = K + 1 : GOTO 12Ø
 16Ø END
1ØØØ DATA   (same as problem)
```

3.
```
4ØØ REM ** SELECT NAME BY PHONE NUMBER
41Ø CLS
42Ø INPUT "PHONE NUMBER" ; N$
43Ø K = 1
44Ø IF NMBR$(K) = N$ PRINT NAYM$(K) : GOTO 42Ø
45Ø IF NMBR$(K) = "ZZZZZ" PRINT "NUMBER NOT FOUND" : GOTO 5ØØ
46Ø K = K + 1 : GOTO 44Ø
5ØØ END
```

4.
```
44Ø IF NMBR$(K) = N$ PRINT NAYM$(K) : GOTO 47Ø
45Ø IF NMBR$(K) = "ZZZZZ" PRINT "NUMBER NOT FOUND" : GOTO 51Ø
46Ø K = K 1 : GOTO 44Ø
47Ø A$ = INKEY$ : IF A$ = "" THEN 46Ø
48Ø IF A$ = "" THEN 44Ø
49Ø IF A$ = "Q" THEN 41Ø
5ØØ GOTO 47Ø
51Ø END
```

5.
```
63Ø A$ = INKEY$ : IF A$ = "" THEN 63Ø
64Ø IF A$ = "" PRINT NAYM$(K+1), NMBR$(K+1)
65Ø IF A$ = "Q" THEN 41Ø
```

CHAPTER SIX

Disk Operation

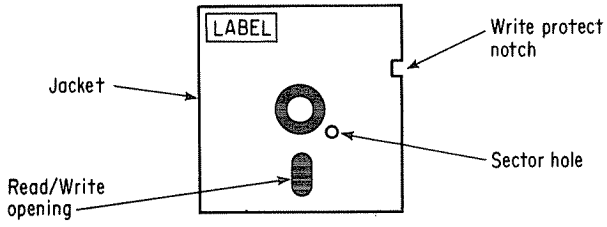
So you want to upgrade your TRS-80 system for disk operation? If you're not sure you need a disk, you should read this chapter to see the capabilities available. If you're not interested in disk operations, skip this chapter and the next one.

In this chapter you will learn:

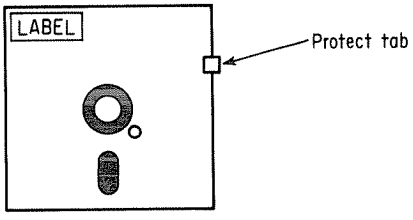
- the advantages of disk over cassette operation,
  - a) faster
  - b) more capacity
  - c) more reliable
- what components are necessary for a minimum TRS-80 disk operating system,
- how to get your disk system running,
- how Radio Shack designates changes to its Disk Operating System by a version number,
- about the care of disks,
- how to make a back-up of the TRSDOS software diskette with a one-disk system,
  - a) the difference between Source and Destination disks
  - b) how to swap disks when prompted
  - c) how a successful back-up is signified
- why a back-up is needed,
- how to display a File Directory,
- how to SAVE a program on disk from BASIC,
- how to LOAD and RUN a program previously stored on disk, and
- how a two-disk system differs from a one-disk system in operation.
  - a) back-up procedure
  - b) disk use

Even if you have a two-drive system, the single-disk procedures will be useful should one of your drives break down.

A diskette is a circular plastic sheet that is coated with a layer of ferromagnetic material. It is permanently sealed inside a protective jacket to prevent bending, creasing, scratching, and contamination from foreign objects. Avoid touching the plastic portion — handle by the jacket carefully.



The write protect notch of the TRSDOS diskette is covered by a tape (or tab) so that the information on it is protected.



The TRS-80 disk drive is a mass storage device providing much more storage than is available on a cassette recorder. Data can be stored on disk and retrieved from disk much quicker than when using a cassette recorder. In addition, the disk is much more reliable and you don't have to worry about critical volume settings, as you do with the cassette recorder.

Read your *TRSDOS and DISK BASIC Reference Manual* (Radio Shack Catalog Number 26-2104) carefully for technical information.

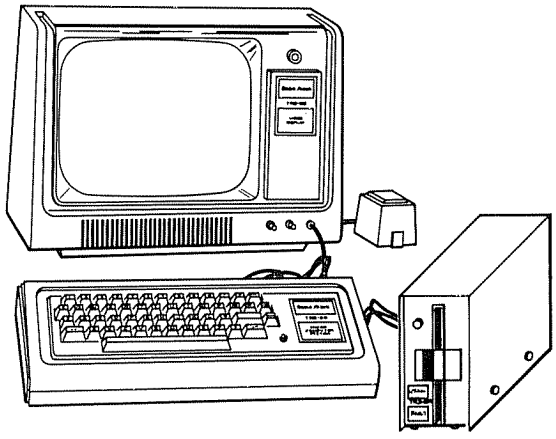
TRSDOS means *Tandy/Radio Shack Disk Operating System*

We will assume that you have learned how to make the necessary connections so that you may get right into disk operation. If you don't remember, study the *TRSDOS and DISK BASIC Reference Manual* that came with your disk system.

In order to run the Radio Shack disk system, you need a minimum of the following:

1) CPU/keyboard unit with 16K RAM,
2) Video monitor,
3) Expansion interface (Radio Shack Catalog Number 26-1140) with 16K additional RAM recommended,
4) One disk drive (Radio Shack Catalog Number 26-1160) and the TRSDOS diskette,
5) One blank diskette (Radio Shack Catalog Number 26-305), and
6) Optional — additional blank diskettes. (Highly recommended!)

The minimum disk system would look like this:



Although a disk system *may* be used with only 16K of RAM, there will be little room left for your own programs. Since one of the primary advantages of a disk system is the capability to store huge programs and lots of data, we recommend that you have a minimum of 32K of RAM. This means 16K in the keyboard unit and 16K more in the expansion interface.

The first disk drive is designated Drive 0 (zero). When the disk system is in use, Drive 0 always contains the TRSDOS diskette, which has the operating system software on it. The diskette also contains an *executive* program to control operations and several auxiliary programs, including DISK BASIC.

Your first disk operation should be to duplicate the TRSDOS diskette onto a *blank* diskette. The disk system is useless without the TRSDOS diskette. Therefore, a duplicate should be made *before* you remove the write protect tape from the TRSDOS diskette. In fact, we recommend that you *never* remove the write protect tab. The back-up diskette which you will make will be a *working* copy of the original diskette.

The write protect tape prevents any information from being *written* onto the diskette. It is placed on the TRSDOS to prevent the vital information already there from being destroyed.

> Duplicate the TRSDOS diskette before you do any other disk operation. Directions for this operation will follow shortly.

Power Up and Duplicate TRSDOS

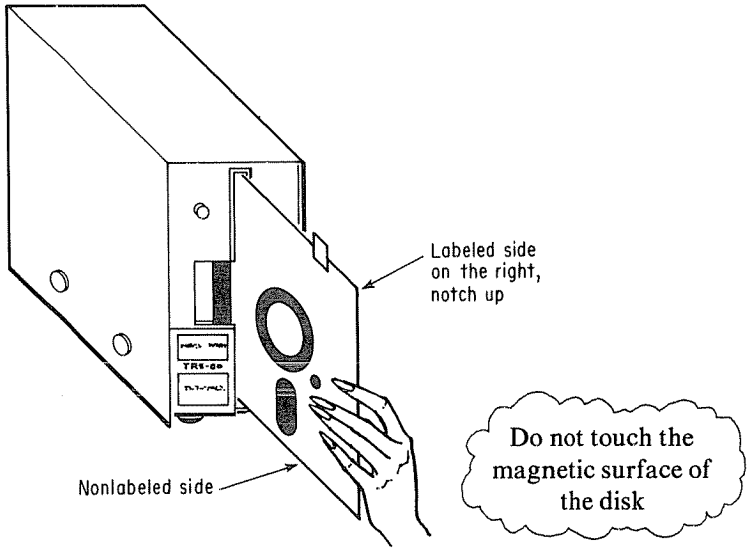
Connect the expansion interface and your disk drive(s) in the manner described in your *TRSDOS and DISK BASIC Reference Manual* (Radio Shack Catalog Number 26-2104).

> Always power up the peripherals (disk drive(s), printer, expansion interface, etc.) first. Turn on your TRS-80 CPU/keyboard unit last of all.

The power switch for the disk drive is on the rear of the unit. Power ON is the up position of the switch, power OFF is down.

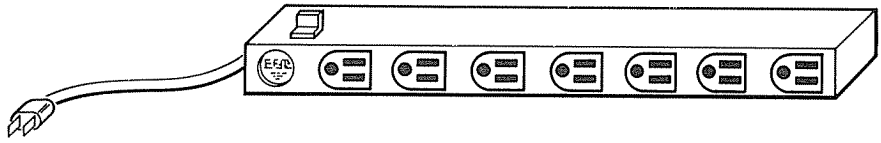
In general,

1) turn on the expansion interface,
2) turn on your disk drive(s),
3) open the front door of the drive and insert the TRSDOS diskette, making sure it is all the way in. Then close the disk door. If it doesn't close easily, don't force it. Reinsert the diskette and try again.



4) Turn on the CPU/keyboard unit. The computer will instantly attempt to load TRSDOS from Drive 0. Therefore, the TRSDOS diskette must be in Drive 0 when the CPU/keyboard is turned on.

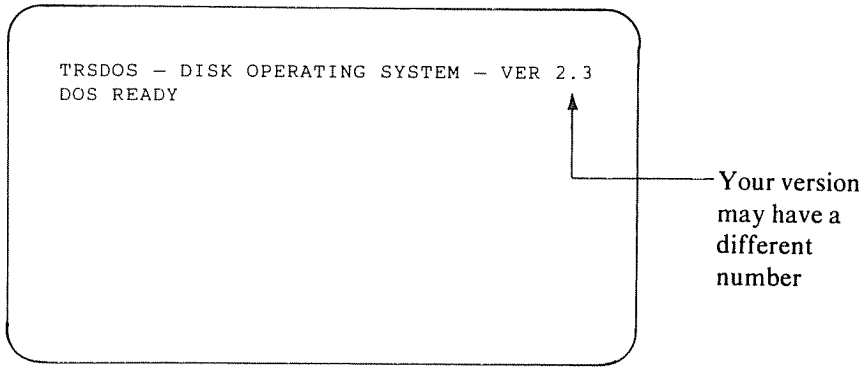
Another approach to these four steps is to plug all devices into an adequate power strip (such as Radio Shack Catalog Number 26-1451) and turn them all on at one time from the power strip's single switch.



Now, power up the TRS-80 for disk operation according to the previous directions, and you'll be ready to duplicate the TRSDOS diskette. Remember, we are assuming that you have only one disk drive (Drive 0).

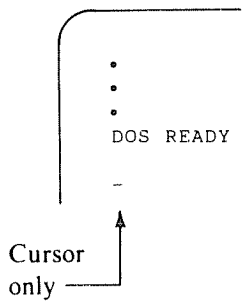
As the system comes on, you'll hear a whirr and clackety-clack as TRSDOS is loaded into memory from the disk. A program called the executive program has been loaded into the first 4K bytes of RAM. It stays there while TRSDOS is in control to make sure the computer will follow the necessary disk operating procedures.

Then the video display shows this message:



The TRSDOS is revised from time to time. A new *version* represents a substantial expansion of the previous version. The integer part of the version number shows which version is present. Ours is version 2. A new *release* is simply an update of the previous version. A later release generally includes wider implementation and enhancements, as well as fixes for earlier problems. A new release is signified by the decimal part of the version number. Ours shows release .3. Thus the complete version number we are using is 2.3.

You can see from the video display that TRSDOS, version 2.3 (or whatever version you have), has been loaded. The prompt "DOS READY" indicates that the disk operating system is ready and waiting for your first command. Notice that the prompt used in BASIC (>) is not displayed. Only the cursor shows.



This can serve as a reminder of whether you are in the disk command mode or BASIC language mode.

Now, it's time to make a back-up copy of the original TRSDOS diskette. A back-up copies the operating system from the original diskette into memory and then onto your blank diskette. When using a one-disk system, each disk that you use must have the operating system on it. Even though programs *may* be stored on the original TRSDOS diskette, it is a good idea to save the original for the back-up procedure only.

You should have both the TRSDOS disk and a blank disk ready now for the back-up procedure.

> When using only one drive, you must swap the TRSDOS diskette (the SOURCE) and your blank disk (the DESTINATION) several times during the back-up procedure. The BACK-UP program will tell you when to insert the DESTINATION disk and when to reinsert the SOURCE disk.

Are you ready to BACK UP your TRSDOS diskette?
If so, . . .

Type: BACKUP (and press ENTER)

Display:

```
TRS DISK BACKUP UTILITY VER 2.3
SOURCE DRIVE NUMBER?-
```

Remember, this procedure is being used with one disk drive (Drive 0). Therefore, . . .

Type: Ø (and press ENTER)

Display:

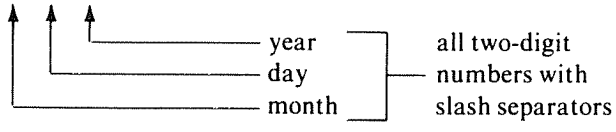
```
TRS DISK BACKUP UTILITY VER 2.3

SOURCE DRIVE NUMBER?Ø
DESTINATION DRIVE NUMBER?-
```

Since there is only one drive, the back-up is made from Drive 0 to Drive 0. That is why the disks must be exchanged in the drive as you go through this procedure. A second drive would make it simpler, but for one drive you must . . .

Type: Ø (and press ENTER)

Display:

```
TRS DISK BACKUP UTILITY VER 2.3

SOURCE DRIVE NUMBER?Ø
DESTINATION DRIVE NUMBER?Ø
BACKUP DATE (MM/DD/YY)
```



Do not type in the parentheses; just add the date that you are making the back-up.

Type: present date

Example: 05/22/81 (for May 22, 1981)

Display:

```
TRS DISK BACKUP UTILITY VER 2.3

SOURCE DRIVE NUMBER?Ø
DESTINATION DRIVE NUMBER?Ø
BACKUP DATE (MM/DD/YY)?Ø5/22/81
```

When you press the ENTER key, TRSDOS will start the BACK-UP procedure. The computer first formats the blank diskette, locking out any defective tracks on the disk. It then duplicates the contents of the TRSDOS diskette onto the blank diskette.

The example shown here is for TRSDOS 2.3. If you have a different version, the number of disk exchanges that are necessary may be different.

```
•
•
•
BACKUP DATE (MM/DD/YY)?Ø5/18/81

INSERT SOURCE DISK (ENTER) <-----
```

this message flashes on and off

If your Source Disk (the TRSDOS diskette) is already in the drive, press ENTER. If not, insert the Source Disk, close the drive door, and press ENTER.

The display shows:

```
•
•
•
•
BACKUP DATE (MM/DD/YY)?Ø5/22/81

INSERT DESTINATION DISK (ENTER) <-----
```

Flashing message

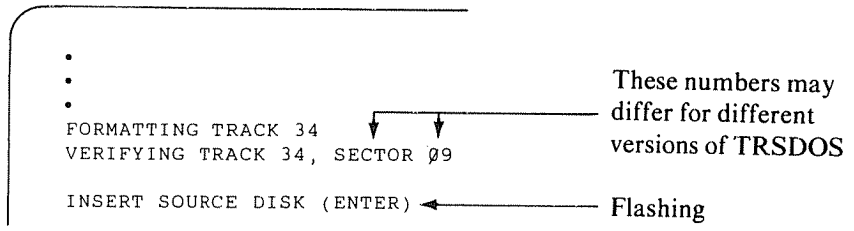
Now, you must make one of the several disk exchanges. Open the drive door, take out the Source Disk and insert the Destination Disk. Then close the drive door and press ENTER.

Then the display shows:

```
•
•
BACKUP DATE (MM/DD/YY)?Ø5/22/81

FORMATTING TRACK 34
VERIFYING TRACK XX, SECTOR YY
```

XX and YY are two-digit numbers that increase as tracks and sectors are examined.

When all tracks and sectors have been verified or locked out, the display shows:



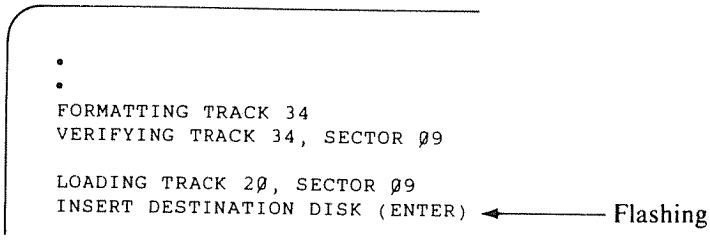
The computer is now ready to copy the necessary information from the TRSDOS disk to your blank disk. Therefore, you have to first place the TRSDOS disk in the drive to load the information from that disk to the computer's memory.

Switch the disks again – Destination out
Source in

Be sure you get the right disks.

and press ENTER

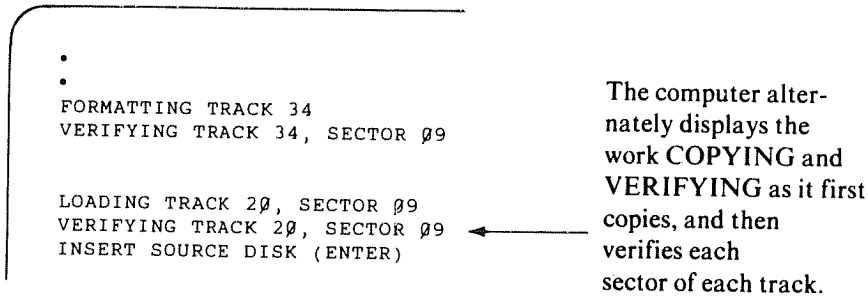
When this portion is finished, the display shows:



Now the computer must copy the data from memory onto the blank disk. Therefore, the blank disk must go into the drive.

Switch disks – Source out
Destination in

Press ENTER and wait . . .



The computer has loaded 20 tracks of the TRSDOS disk, copied them onto the blank disk, and verified that all 20 tracks were copied correctly. It's now time to load some more information from the TRSDOS disk.

Switch disks again and press ENTER.

```
.
.
FORMATTING TRACK 34
VERIFYING TRACK 34, SECTOR Ø9

LOADING TRACK 2Ø, SECTOR Ø9
VERIFYING TRACK 2Ø, SECTOR Ø9
LOADING TRACK 31, SECTOR Ø9
INSERT DESTINATION DISK (ENTER)  <-------- Flashing
```

Tracks 21 through 31 have been loaded from the TRSDOS disk into the computer's memory. It's now time to copy them onto the blank disk.

Switch disks again – destination disk in – then press ENTER.

```
.
.
LOADING TRACK 2Ø, SECTOR Ø9
VERIFYING TRACK 2Ø, SECTOR Ø9
LOADING TRACK 31, SECTOR Ø9
VERIFYING TRACK 31, SECTOR Ø9  <-------- COPYING then
INSERT SOURCE DISK (ENTER)                VERIFYING
```

Tracks 21 through 31 have now been copied and verified. Just a little more to go. More data must now be loaded from the Source Disk.

Switch disks again, press ENTER.

```
.
.
LOADING TRACK 2Ø, SECTOR Ø9
VERIFYING TRACK 2Ø, SECTOR Ø9 .
LOADING TRACK 31, SECTOR Ø9
VERIFYING TRACK 31, SECTOR Ø9
LOADING TRACK 34, SECTOR Ø9  <-------- Almost done
INSERT DESTINATION DISK (ENTER)
```

Tracks 32 through 34 are loaded. They must now be copied and verified. You need the blank disk (Destination) in the drive again.

Switch disks one more time and press ENTER.

```
.
.
LOADING TRACK 2Ø, SECTOR Ø9
VERIFYING TRACK 2Ø, SECTOR Ø9
LOADING TRACK 31, SECTOR Ø9
VERIFYING TRACK 31, SECTOR Ø9
LOADING TRACK 34, SECTOR Ø9
VERIFYING TRACK 34, SECTOR Ø9
BACKUP COMPLETE  <-------------- DONE AT LAST

HIT 'ENTER' TO CONTINUE
```

Watch for the BACKUP COMPLETE. If your disk BACK-UP has been successful, that is the message you will see at the end of the operation. However, if you see:

```
.
.
.
BACKUP REJECTED DUE TO (. . . . .)
```

erase the diskette with a bulk eraser (such as Radio Shack Catalog Number 44-210) and repeat the BACK-UP procedure. If it still doesn't work, try using another blank diskette.

> NOTICE: The BACK-UP utility program is provided by Radio Shack *solely* for *your* personal use. It is legal to use it to maintain copies of *your* TRSDOS and data diskettes. It is not legal to make copies for sale.

You may now put your original TRSDOS disk away for safekeeping. Keep it in a protective envelope. A case for protecting up to 10 disks is available from Radio Shack (Catalog Number 26-1452). Radio Shack also has a box which will protect up to fifty disks (Catalog Number 26-1450).

From now on, use the working copy. (Hmm . . . then it is not a back-up copy.)

Displaying Disk Files

You now have a back-up copy of the TRSDOS on your previously blank disk. Use this as a working disk in the future and save the original TRSDOS disk for making future back-ups.

The disk operating software must be in the system for disk operation. Therefore, if you have a one-disk system, TRSDOS should be available on the disk that you are using.

> Each time you need a new disk, repeat the BACK-UP procedure to copy TRSDOS from the original disk to the new blank disk.

To see what has been copied onto your working disk, place it in the disk drive and close the drive door. If you continue from the completed BACK-UP procedure, it is already in the drive. After the "HIT 'ENTER' TO CONTINUE" message, do as it says — press the ENTER key.

```
TRSDOS - DISK OPERATING SYSTEM - VER 2.3

DOS READY
-
```

Whenever the DOS READY prompt is displayed, you may enter a command.

To see what programs have been placed on your disk, look at the file directory.

Type: DIR, and press ENTER

```
FILE DIRECTORY --- DRIVE Ø      TRSDOS -- Ø5/22/81

TEST1/CMD          GETDISK/BAS    TEST2/BAS
DISKDUMP/BAS       GETTAPE/BAS    TAPEDISK/CMD


DOS READY

-
```

The files shown in the directory are utility programs available for your use. Notice the letters following the slash in each file name. CMD means the program is accessed by a command from TRSDOS. BAS means that DISK BASIC must be accessed before using the program.

TRSDOS provides many of these utility programs that you will use as you become more familiar with the disk system. A detailed discussion of the utilities would be inappropriate at this time. However, you will be using the DIRectory to look for your own programs. You will come back to the directory after you have SAVEd a program of your own.

Saving a BASIC Program on Disk

One of your first disk operations is to save a BASIC program on your disk. To demonstrate, let's use the short program from chapter 3 that painted the screen white and poked black holes in the painting.

PAINT SCREEN AND POKE HOLES

```
10 REM * LIGHT A WHOLE BLOCK *
20 CLS
30 FOR M = 15360 TO 16383
40   POKE M,191
50 NEXT M
60 REM * POKE BLACK HOLES *
70 R = RND(1024)+15359
80 POKE R,128
90 GOTO 70
```

To enter the program with the disk system on, you must first access DISK BASIC.

```
.
.
.
DOS READY

BASIC-
```


Type: BASIC (and press ENTER)

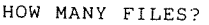
```
HOW MANY FILES?-
```

You respond to this question with the maximum number of disk files that will be in use (such files are said to be *open*) at any one time. The number must be from 0 through 15. If no number is given, 3 files will be open for use. Rather than go into a lengthy discussion of file space at this point, we'll not input a number. Just press the ENTER key.

```
HOW MANY FILES?
MEMORY SIZE? -
```

The second question should be familiar to you from Level II BASIC. Since you are not protecting any memory for machine language programs, just press ENTER again.

```
HOW MANY FILES?
MEMORY SIZE?
RADIO SHACK DISK BASIC VERSION 2.2
READY
>-
```



prompt and cursor

Now enter the PAINT SCREEN AND POKE HOLES program, lines 10 through 90.

```
>10 REM * LIGHT A WHOLE BLOCK *
.
.
.
.
>80 POKE M,128
>90 GOTO 70
>-
```

To save the program, which is now in the computer's memory type:

SAVE "HOLES/BAS"

Name of program — for a BASIC program

This command (SAVE "HOLES/BAS") saves program in *compressed* format, which takes up less disk space and is faster loading and saving than other formats.

Press the ENTER key after the SAVE instruction. The program is copied from the computer's memory to disk. When it is finished (it won't take long), the display will show:

```
.
.
.
>90 GOTO 70
>SAVE "HOLES/BAS"
READY
>-
```

Notice that both prompt and cursor are displayed; this means that you are still in BASIC.

To get back to TRSDOS at any time . . .

Type: CMD "S" and press ENTER

You should go back to TRSDOS to make sure that the program has really been saved. Use CMD "S" to do this.

```
>9Ø GOTO 7Ø
>SAVE "HOLES/BAS"
READY
>CMD "S"

DOS READY <-------- Now you are back in TRSDOS.
```

Now look at the directory to see if the program is there.

Type: DIR, and press ENTER

```
FILE DIRECTORY --- DRIVE Ø   TRSDOS -- Ø5/22/81

TESTI/CMD   GETDISK/BAS   TEST2/BAS
HOLES/BAS   DISKDUMP/BAS   GETTAPE/BAS
TAPEDISK/CMD

DOS READY

_
```

There's your program

Let's go back to BASIC now to see if we can access the HOLES program from the disk. Remember the necessary command?

```
.
.
.
DOS READY

BASIC-          (then press ENTER)
```

```
HOW MANY FILES? <-------- (press ENTER)
MEMORY SIZE? <---------- (press ENTER)
RADIO SHACK DISK BASIC VERSION 2.3
READY
>_
```

Since you are in BASIC, you should LIST to see if the HOLES program is still in memory.

```
.
.
READY
>LIST
READY
>-
```

Going back to DOS seems to have wiped the program out of memory.

Since the program is no longer in memory, you'll have to load it back in from the DISK. Oh well, that's what you really wanted to learn how to do anyway. At this point, you may either LOAD the program from disk into memory and then RUN it, or you may LOAD and RUN it immediately with one command. We'll show both methods.

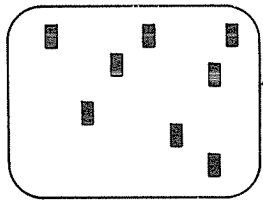
a) First method – LOAD program, then RUN it.

Type: LOAD "HOLES/BAS" (and press ENTER).

```
.
.
READY
>LOAD "HOLES/BAS"
READY
>-
```

After a brief whirr, the program is loaded.

Type: RUN (and press ENTER).



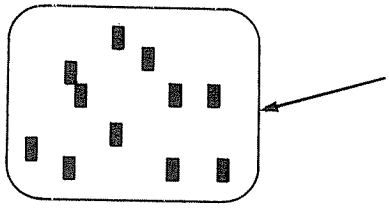
Screen is painted white and black holes appear — the program is running.

Press the BREAK key to stop the program.

b) Second method – LOAD and RUN immediately.

Type: NEW to erase the program from memory.

Then type: RUN "HOLES/BAS" (and press ENTER).



Again, press the BREAK key to stop the program.

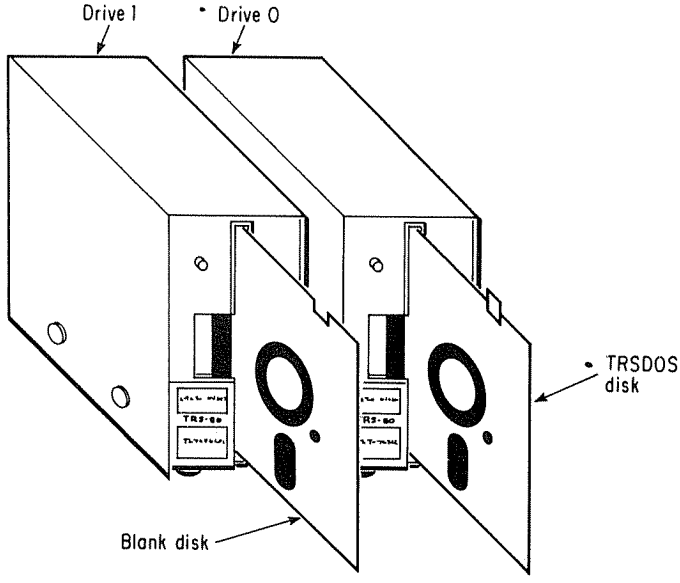
You have now used your disk to save your first BASIC program. You can find it, load it, and run it at any time. Be sure to label any disks on which you have placed programs so that you can quickly find them. If you don't, you'll have to search the directories of each of your disks until you find the program that you want.

Using a Two-Disk System

If you have two disk drives, you will find that your system is more convenient and easier to use than a one-drive system. One drive (Drive 0) can be used to hold a disk that has the operating software on it. The second drive (Drive 1) can then hold a "working" disk that does not need to contain the operating system software. Therefore, there will be much more storage space on the disk for your own files.

We'll show you how easy it is to make a back-up of the TRSDOS disk with a two-disk system.

1) Insert the TRSDOS disk in Drive 0.
2) Insert a blank disk in Drive 1.



3) Power up your system and go.

```
TRSDOS - DISK OPERATING SYSTEM - VER 2.3
DOS READY

-
```

Type: BACKUP (and press ENTER).

```
TRS DISK BACKUP UTILITY VER 2.3
SOURCE DRIVE NUMBER?-
```

The Source disk (TRSDOS) is in Drive 0, so . . .
Type: 0 (and press ENTER).

```
TRS DISK BACKUP UTILITY VER 2.3

SOURCE DRIVE NUMBER?Ø
DESTINATION DRIVE NUMBER?-
```

Your Destination disk (now blank) is in Drive 1, so . . .
Type: 1 (and press ENTER).

```
TRS DISK BACKUP UTILITY VER 2.3

SOURCE DRIVE NUMBER?Ø
DESTINATION DRIVE NUMBER?1
BACKUP DATE (MM/DD/YY)?-
```

Type: the present date (say May 28, 1981).

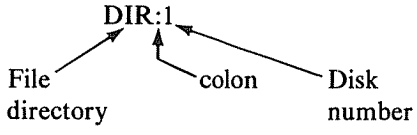
```
.
.
SOURCE DRIVE NUMBER?Ø
DESTINATION DRIVE NUMBER?1
BACKUP DATE (MM/DD/YY)?Ø5/28/81-
```

When you press the ENTER key, the back-up procedure begins. With two disks, you no longer have to swap disks as the process goes along. The back-up goes merrily on its way uninterrupted.

```
.
.
.
.
BACKUP DATE (MM/DD/YY)?Ø5/28/81
FORMATING TRACK 34
VERIFYING TRACK 34, SECTOR Ø9
VERIFYING TRACK 34, SECTOR Ø9
VERIFYING TRACK 34, SECTOR Ø9
BACKUP COMPLETE


HIT 'ENTER' TO CONTINUE
```

This goes on continuously until the sectors of all tracks are loaded, copied, and verified.

The process is much quicker and easier if you have two disk drives. To see the file directory of the disk system in Drive 1, you must specify the drive number.



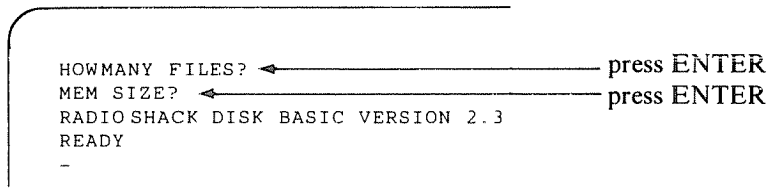
- The command DIR will display the file directory of the disk in Drive 0.
- The command DIR:1 will display the file directory of the disk in Drive 1.

LOAD, SAVE, and RUN with a Two-Disk System

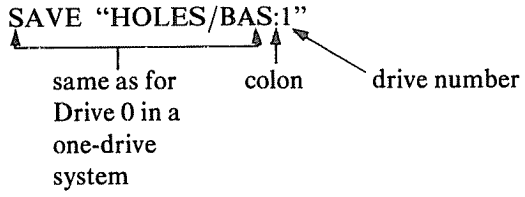
The commands to SAVE, LOAD, and RUN BASIC programs are similar to those demonstrated with the one-disk system. Once again, the disk drive number must be specified if the drive desired is not Drive 0. To save a program from the computer's memory to the disk in Drive 1, first access DISK BASIC.

```
.
.
.
DOS READY

BASIC-
```

Type: BASIC and press ENTER



Type:



The same is true to LOAD or RUN a BASIC program that is on Drive 1.

LOAD "HOLES/BAS:1"
or
RUN "HOLES/BAS:1"

As stated earlier, the disk in Drive 1 does not have to contain the disk operating software (it is available from Drive 0). Therefore, more storage space is available on the second disk.

Summary

You now have some knowledge of TRSDOS (Tandy/Radio Shack Disk Operating System). Although, you have barely scratched the surface of the disk operating system, you have made a good beginning. You learned:

- that disk operation is faster, more reliable, and has more storage capacity than cassette operation,
- that an expansion interface, 32K of memory, and at least one disk drive are needed,
- a new power-up procedure,
- that the first drive in your system is named Drive 0, and it must contain the TRSDOS software disk,
- that TRSDOS has been published in several versions — version 2.3 is demonstrated in this chapter,
- to BACK UP your TRSDOS diskette with one disk drive,
    a) The Source disk contains TRSDOS.
    b) The Destination Disk is originally blank — the BACK-UP copies TRSDOS from the Source disk onto the Destination disk.
- to display the file directory of the disk in Drive 0,

- to SAVE a BASIC program from memory onto Drive 0,

```
SAVE "HOLES/BAS"
      ↗        ↖
file name    a BASIC program
```

- to LOAD, or LOAD and RUN a BASIC program from disk,

```
LOAD "HOLES/BAS" ←—— loads a BASIC program
RUN "HOLES/BAS" ←—— loads and runs a BASIC
                    program
```

- how to get back and forth between TRSDOS and BASIC,

```
BASIC ←———————— a TRSDOS command that executes DISK BASIC
CMD "S" ←———————— a BASIC command that sends control back
                   to TRSDOS
```

- how easy it is to BACK UP TRSDOS using a two disk system, and
- to alter the SAVE, LOAD, and RUN commands for use on a two disk system.

```
SAVE "HOLES/BAS:1" ←—saves "HOLES" on disk Drive 1
LOAD "HOLES/BAS:1" ←—loads "HOLES" from disk Drive 1
RUN "HOLES/BAS:1" ←—loads and runs "HOLES" from
                    disk Drive 1
```

Self-Test

1. What do the letters in "TRSDOS" represent?

   _____

2. Why must TRSDOS be on each disk if you are using a one-drive system?

   _____

   _____

3. What is the purpose of a write protect tape on a disk?

   _____

   _____

4. Number the power-up steps in the recommended sequence (1,2,3,4).

   _____ Turn on the keyboard unit
   _____ Turn on the expansion interface
   _____ Insert disk
   _____ Turn on disk drive(s)

5. This chapter demonstrates TRSDOS version 2.3 which means:

   _____ version (2 or 3?) _____ release (2 or 3?)

6. Tell in your own words why you have to swap disks when performing a BACK-UP with a one-drive system.

   _____

   _____

   _____

   _____

7. When starting the BACK-UP procedure, TRSDOS is on the _____ disk.
(source, destination)
8. When using a one-drive system, give the correct response to these questions.

SOURCE DRIVE NUMBER?_____
DESTINATION DRIVE NUMBER? _____
9. Write in the correct command for displaying the file directory of the disk in Drive 0.

_____
10. In order to SAVE a BASIC program named "WHITE" on a disk with a one-drive system, the correct command would be:

_____
11. If you are using DISK BASIC, and CMD "S" is executed, fill in the display.

```
.
.
.
READY
>CMD'"S"
```

12. Why is it easier to make a BACK-UP with a two-drive system than with a one-drive system?

_____
13. If you have two drives, what would be your answers to the following?

```
TRS DISK BACKUP UTILITY VER 2.3

SOURCE DRIVE NUMBER?_____

DESTINATION DRIVE NUMBER?_____
```

14. Suppose you have a BASIC program named "COUNT" on a disk in Drive 1, and you want to load and run it immediately. Give the necessary DISK BASIC command.

_____

Answers to Self-Test

1. Tandy/Radio Shack Disk Operating System
2. Because the operating software contained in TRSDOS must be available for successful disk operation
3. So that information cannot be written over the essential TRSDOS software
4. 4 Turn on the keyboard unit
   1 Turn on the expansion interface
   3 Insert disk
   2 Turn on disk drive(s)
5. 2 version
   3 release
6. The BACK-UP procedure loads information from TRSDOS (the Source) and then must copy it back to a blank disk (the Destination). Therefore, disks must be exchanged.
7. Source
8. SOURCE DRIVE NUMBER?0
   DESTINATION DRIVE NUMBER?0
9. `DIR  (or DIR:Ø)`
10. `SAVE "WHITE/BAS"`
11. `DOS READY`
    `_`
12. The disks do not have to be swapped.
13. `SOURCE DRIVE NUMBER?Ø`
    `DESTINATION DRIVE NUMBER?1`
14. `RUN "COUNT/BAS:1"`

CHAPTER SEVEN

Using Disk Files

Now that you know how to get the disk system running, we will explore the use of sequential disk files. In this chapter you will learn:

- what the computer wants when it asks "HOW MANY FILES?" as you access Disk BASIC from DOS,
- the format to OPEN disk files for use,
- the format to CLOSE disk files after you have finished using the files,
- how to use a program to create a disk file and save it on a disk,
- how to use a program to load a disk file from a disk into the computer's memory,
- how to add data to a disk file, and
- how to use a program that includes the ability to:
  a) create a new data file from the keyboard,
  b) input a data file from disk,
  c) add records to a data file,
  d) delete records from a data file, and
  e) examine records in a data file.

Creating a Data File

In chapter 6 you learned how to save a BASIC program in a disk file. You will now learn to create a data file. A data file is not a program, but merely a list of information that you might want to save and use at a later date. You know how to save data on cassette tape. Disk data files are not much different.

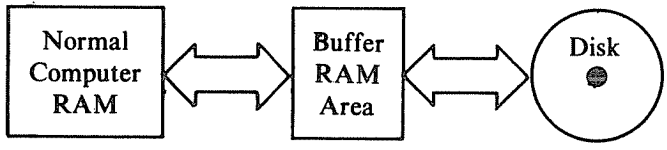
After you turn on your disk system and see the DOS READY prompt, you access disk BASIC to create a data file.

```
.
.
.
DOS READY
BASIC-
```
← Type BASIC and press ENTER

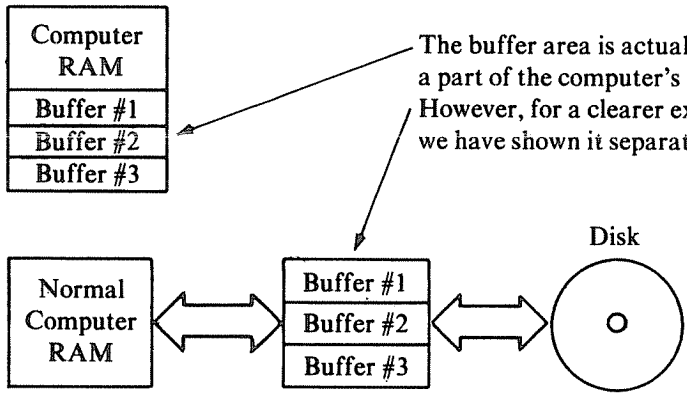
The first thing that the computer wants to know is how many files you will be using.

```
HOW MANY FILES?-
```

The number that you type in tells the computer how many *buffers* to create to handle the disk accesses you will make (the number of READs and WRITEs). A buffer is an area of memory where datum waits when it is going to and from the disk file.

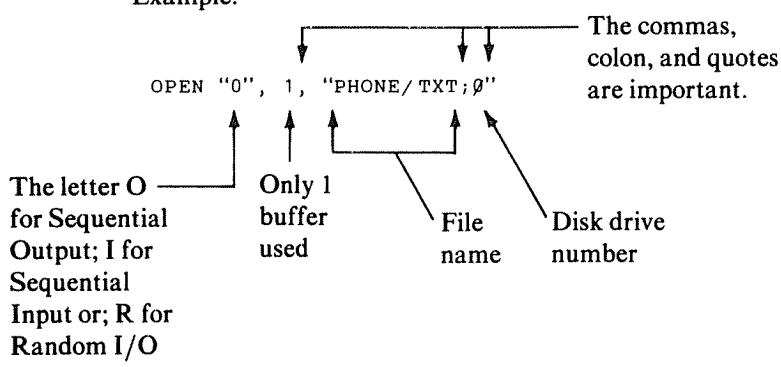


If you do not give a number (but merely press the ENTER key) the computer automatically reserves three buffers. In most cases, you will not use more than three buffers. Therefore, you can just press the ENTER key to go on.



When you want to access a particular disk file, you must tell the computer which buffer to use to access the file. You must also tell it what kind of access you want (Sequential Output, Sequential Input, or Random I/O). All this is done in one statement — OPEN — and undone with another statement — CLOSE.

Example:



You need a BASIC program to input the data into the computer's memory and to send it to the disk. Here is a short program to do that. Our numbering system may seem odd, but the reasons behind it will appear obvious later on.

Program to Create a New Data File

```
 1Ø REM * HOUSEKEEPING *
 2Ø CLEAR 5ØØ
 3Ø DIM A$(3Ø),B$(3Ø)
 4Ø CLS

1ØØ REM * CREATE A FILE *
11Ø INPUT "HOW MANY NAMES";R
12Ø FOR N = 1 TO R
13Ø   INPUT "NAME";A$(N)
14Ø   INPUT"PHONE # (XXX-XXXX)";B$(N)
15Ø NEXT N
16Ø LINE INPUT "FILE NAME? (XXXXXXX/TXT:Ø)";D$
17Ø PRINT "PRESS 'ENTER'"
18Ø INPUT "TO SEND TO DISK";C$

19Ø REM * MEMORY TO DISK *
2ØØ OPEN "O",1, D$
21Ø PRINT#1,R
22Ø FOR N = 1 TO R
23Ø   PRINT#1,A$(N);",";B$(N)
24Ø NEXT N
25Ø CLOSE
26Ø END
```

Clear string space and Dimension arrays

Input data to memory

Open file

Send data to disk

Close the file

Note: A new disk BASIC statement is used in line 160. LINE INPUT is a statement that will allow a string input that includes commas, quotes, and other punctuation marks.

Also note line 250:

DO NOT REMOVE A DISK WHICH CONTAINS AN OPEN FILE. * CLOSE THE FILE FIRST *

Before you try the file program, be sure to save it on disk. You may want to use it again sometime. Remember the command to save a BASIC program?

```
HOW MANY FILES?          <-- You pressed ENTER
MEM SIZE?                <-- You pressed ENTER and the
.                            computer went to BASIC
.
>25Ø CLOSE               <-- Type in the BASIC program
>26Ø END
>SAVE "PHONE/BAS:Ø"      <-- This saves your BASIC
READY                        Input program on disk
>-
```

Suppose that you want to create a data file of names and phone numbers of a club, your friends, or some other group. Here is a list of five names and phone numbers to be entered. You should start out with a small list. You can learn to expand or alter the list later.

Name Last, First initial	Phone Numbers
Able B	555–0101
Baker D	555–1010
Candy K	555–1111
Dunks C	333–2020
Dunks D	333–2020

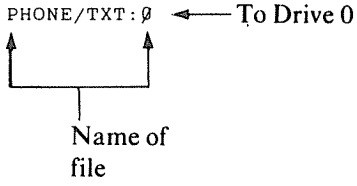
Run the program and input the names and phone numbers in response to the input prompts.

```
.
.
READY
>RUN
HOW MANY NAMES?5          <-- Five names and phone numbers
NAME?ABLE B               <-- First name
PHONE # (XXX-XXXX)?555-Ø1Ø1   <-- First phone number
NAME?-                    <-- Ready for second name
```

After the last phone number has been typed in, the computer displays the entries:

```
ABLE B     555-Ø1Ø1
BAKER D    555-1Ø1Ø
CANDY K    555-1111
DUNKS C    333-2Ø2Ø
DUNKS D    333-2Ø2Ø
FILE NAME? (XXXXXXX/TXT:Ø)-
```

The program is now requesting you to enter the name of the data file. We will call this one:



The word PHONE is typed in in place of the X's in the request for file name. This is not a BASIC file so we used TXT for the type of file. When you type in the file name it is assigned to the variable D$. When you type in the file name and press ENTER, the computer then gives another prompt message and waits for you to press a key before sending the data to the disk.

```
FILE NAME? (XXXXXXX/TXT:Ø) PHONE/TXT:Ø
PRESS ENTER
TO SEND TO DISK?-                      <-- You type this
```

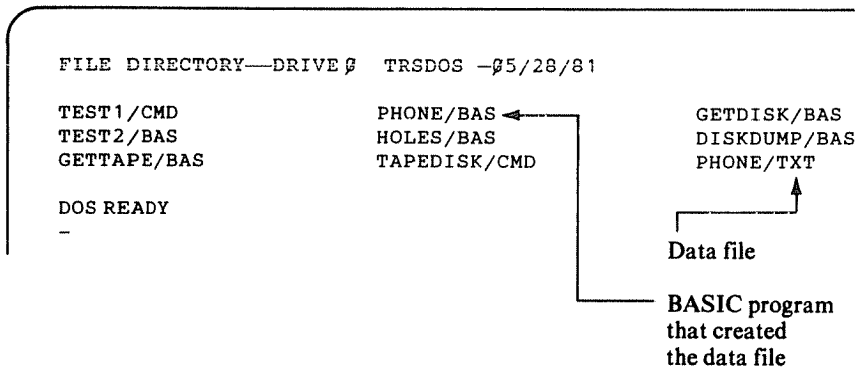
Press ENTER and the data will be saved on disk under the file name PHONE/TXT:0.

Then:

```
PRESS 'ENTER'
TO SEND TO DISK <-- The data file is saved on the disk
READY
>-
```

You will no doubt want to check the disk directory to make sure your original BASIC program and the data file are actually on the disk. You must go back to the disk operating system to do this. Just type: CMD "S" and then type DIR.

```
.
.
READY
>CMD "S"- <-- Type and press ENTER
```



Since both programs are there, you have successfully passed the first test. However, at some time your list of names and phone numbers has expanded to nineteen, as shown below.

Name	Phone number
Able B	555-0101
Baker D	555-1010
Candy K	555-1111
Dunks C	333-2020
Dunks D	333-2020
Evers L	555-2130
Fink F	333-1212
Good K	555-3120
Henry B	555-2212
Irving D	333-1115
Jones E	555-7632
Kane K	555-7603
Lemon R	333-1515
Marks H	555-8119
Noble M	333-8991
Otto D	333-1129
Pinks C	555-3210
Riot C	555-4567
Sample F	333-7654

From original list

To be added

You must now make a decision. Should you go back and use the PHONE/BAS input program and input the whole list over, or should you write a new program to change the data in the data file? Since you only have five names in the data file named PHONE/TXT, you might be tempted to just do it all over again. However, your list may grow later, and you may have to add only a few names. It would probably be to your advantage to write a new program that could add names to the list.

First you need a BASIC program to load the data file back into the computer. Remember the original input program PHONE/BAS opened a file for *output*. Since you want to input data from the disk, you must open the file for Sequential Input.

```
OPEN "I",1, "PHONE/TXT:Ø"
```

Open for Sequential Input

That's the file you want

Program to Input File from Disk

```
 1Ø REM * HOUSEKEEPING *
 2Ø CLEAR 5ØØ
 3Ø DIM A$(3Ø),B$(3Ø)
 4Ø CLS

3ØØ REM * INPUT DATA FILE *
31Ø LINE INPUT "NAME OF FILE?";D$
32Ø OPEN "I",1,D$
33Ø INPUT#1,R
34Ø CLS
35Ø FOR N = 1 TO R
36Ø   INPUT#1,A$(N),B$(N)
37Ø   PRINT A$(N),B$(N)
38Ø NEXT N
39Ø CLOSE
4ØØ INPUT "PRESS 'ENTER' TO CONTINUE";C$
41Ø END
```

Don't forget the CLOSE statement

Access disk BASIC before entering the program.

```
DOS READY
BASIC
READY
>-
```

Enter the program and run it. The disk whirrs and merrily clickety-clacks as it finds the PHONE/TXT file and loads and prints the old names and telephone numbers.

```
ABLE B     555-Ø1Ø1
BAKER D    555-1Ø1Ø
CANDY K    555-1111
DUNKS C    333-2Ø2Ø
DUNKS D    333-2Ø2Ø
READY
>-
```

You need a program that will add fourteen more names to the data file. To make it more general, you should be able to input where you want to start adding names and where you want to stop. You don't want to disturb the five names that are already in the file.

Since you will usually want to input the names and phone numbers that you already have, you might as well add this program to the program that inputs the data file.

```
5ØØ REM * ADD RECORDS *
51Ø INPUT "ADD HOW MANY RECORDS";A
52Ø FOR N = R+1 TO R+A
53Ø   INPUT "NAME";A$(N)
54Ø   INPUT "PHONE # (XXX-XXXX)";B$(N)
55Ø NEXT N
56Ø REM * SEND ALL TO DISK *
57Ø OPEN "O",1,D$
58Ø PRINT#1,R+A
59Ø FOR N = 1 TO R+A
6ØØ   PRINT#1,A$(N);",";B$(N)
61Ø NEXT N
62Ø CLOSE
63Ø END
```

Number of new names to be added

Same name will erase old file

New data put in file (old names plus new)

Add lines 500 through 630 to lines 10 through 410 of the Program to Input File from Disk. Then SAVE it for the future as PHLOAD/BAS:0 (for PHone LOAD from disk 0).

```
.
.
.
>62Ø CLOSE
>63Ø END
>SAVE "PHLOAD/BAS:Ø"
READY
>-
```

Now run the BASIC program again and input the additions to the list.

You may see a need for some organization as your list of programs to handle this data changes. You can now add names and numbers to the list, but pretty soon you will want to do additional things with the file. You should plan such things in advance and combine the separate programs with a Menu Selection to perform the desired results. For example, you might want to do these things:

1. Create a new data file.
2. Input a data file from disk.
3. Add records to a data file.
4. Delete records from a data file.
5. Examine the records in a data file.

You have worked with the first three things on the list. Now let's combine them all into one program with a menu to select the program you want.

The first item on the list was numbered with lines 100 through 260. The second used lines 300 through 400. The third used lines 500 through 630. With some planning, we have avoided any line number conflicts. All you have to do is put the five programs together and plan a separate Menu Selection section.

Data File Program

```
 1Ø REM * HOUSEKEEPING *
 2Ø CLEAR 5ØØ
 3Ø DIM A$(3Ø),B$(3Ø)
 4Ø CLS
 5Ø GOTO 5ØØØ <-------------------- 5000 lists the menu

1ØØ REM * CREATE A FILE *
11Ø INPUT "HOW MANY NAMES";R
12Ø FOR N = 1 TO R
13Ø   INPUT "NAME";A$(N)
14Ø   INPUT "PHONE # (XXX-XXXX)" :B$(N)
15Ø NEXT N
16Ø LINE INPUT "FILE NAME? (XXXXXXX/TXT:Ø)";D$
17Ø PRINT "PRESS 'ENTER'"
18Ø INPUT "TO SEND TO DISK";C$
19Ø REM * MEMORY TO DISK *
2ØØ OPEN "O"1,D$
21Ø PRINT #1,R
22Ø FOR N = 1 TO R
23Ø   PRINT #1,A$(N);",";B/$(N)
24Ø NEXT N
25Ø CLOSE
26Ø GOTO 5ØØØ <-------------------- Go to menu

3ØØ REM * INPUT FILE FROM DISK *
31Ø LINE INPUT "NAME OF FILE?";D$
32Ø OPEN "I",1,D$
33Ø INPUT #1,R
34Ø CLS
35Ø FOR N = 1 TO R
36Ø   INPUT#1,A$(N),B$(N)
37Ø   PRINT A$(N),B$(N)
38Ø NEXT N
39Ø CLOSE
4ØØ INPUT "PRESS 'ENTER' TO CONTINUE";C$
41Ø GOTO 5ØØØ
```

```
500 REM * ADD RECORDS *
510 INPUT "ADD HOW MANY RECORDS";A
520 FOR N = R+1 TO R+A
530   INPUT "NAME";A$(N)
540   INPUT "PHONE # (XXX-XXXX)";B$(N)
550 NEXT N
560 REM * SEND ALL TO DISK *
570 OPEN "O",1,D$
580 PRINT#1,R+A
590 FOR N = 1 TO R+A
600   PRINT#1,A$(N);",";B$(N)
610 NEXT N
620 CLOSE
630 GOTO 5000

700 REM * DELETE RECORDS *
710 PRINT "NOT IMPLEMENTED YET"
720 FOR X = 1 TO 500:NEXT X
730 GOTO 5000

900 REM * EXAMINE RECORDS *
910 PRINT "NOT IMPLEMENTED YET"
920 FOR X = 1 TO 500:NEXT X
930 GOTO 5000

1100 REM * END OF PROGRAM
1110 END

5000 REM * MENU SELECTION *
5010 CLS
5020 PRINT @86,"PHONE DIRECTORY MENU"
5030 PRINT @214,"1. CREATE A NEW FILE"
5040 PRINT @278,"2. INPUT FILE FROM DISK"
5050 PRINT @342,"3. ADD RECORD(S)"
5060 PRINT @406,"4. DELETE RECORD(S)"
5070 PRINT @470,"5. EXAMINE RECORD(S)"
5080 PRINT @534,"6. QUIT"
5090 PRINT @662,"TYPE NUMBER OF SELECTION (1-6)";
5100 INPUT N: N=INT(N)
5110 IF N>=1 AND N<=6 THEN 5140
5120 PRINT @790,"MUST BE A NUMBER 1-6"
5130 PRINT @662,STRING$(40," "):GOTO 5090
5140 ON N GOTO 100,300,500,700,900,1100
```

Items 4 and 5 on the menu haven't been written yet, but they can now be added when ready. Before you do anything else, though, enter the Data File Program and SAVE it AS: "DATFILE/BAS:0."

```
.
.
.
5140 ON N GOTO 100,300,500,700,900,1100
SAVE "DATFILE/BAS:0"-
```

The :0 is optional. It is used to indicate that we are using Drive 0.

Now take another look at the directory to see what programs we now have on disk. Some of them should be removed since DATFILE/BAS performs the same operation. Remember how to access the directory?

```
.
.
.
>514Ø ON N GOTO 1ØØ,3ØØ,5ØØ,7ØØ,9ØØ,11ØØ
SAVE "DATFILE/BAS:Ø"
READY
>CMD "S"-
```

Then type: DIR

```
FILE DIRECTORY ——DRIVE Ø    TRSDOS    ——Ø5/28/81

TEST1/CMD          PHONE/BAS          GETDISK/BAS
TEST2/BAS          HOLES/BAS          DISKDUMP/BAS
GETTAPE/BAS        TAPEDISK/CMD       PHONE/TXT
PHLOAD/BAS         DATFILE/BAS

DOS READY
-
```

You won't need these anymore. DATFILE/BAS does the same thing and will do more later.

To erase the old files that are not needed, go back to Disk BASIC and use the KILL command.

```
DOS READY
BASIC <———————— Type: BASIC and press ENTER

READY

>KILL "PHONE/BAS" <———— Type: KILL "PHONE/BAS" and press ENTER
READY
>KILL "PHLOAD/BAS" <———— Type: KILL "PHLOAD/BAS" and press ENTER
READY
>-
```

Take one more look at the directory to see if everything is as planned.

```
READY
>CMD "S"- <———— Type: CMD "S" and press ENTER
```

```
.
.
DOS READY
DIR-
```

Type: DIR and press ENTER

```
FILE DIRECTORY --DRIVE 0  TRSDOS  --05/28/81

TEST1/CMD        GETDISK/BAS      DATFILE/BAS
TEST2/BAS        HOLES/BAS        DISKDUMP/BAS
GETDISK/BAS      TAPEDISK/BAS     PHONE/TXT

DOS READY
-
```

Your two programs; BASIC and data

You now have a program, DATFILE/BAS, that can be used to 1) create a new disk data file, 2) load an existing data file, and 3) add records to an existing data file. Two other operations still to be written are 4) delete records from an existing data file, and 5) examine all or part of an existing data file. You have space (lines 700 through 1000) for these sections to be added. Here is the section to delete records.

Program Section to Delete Records

```
700 REM * DELETE RECORDS *
710 A=0: B=0
720 INPUT "NAME TO BE DELETED";N$
730 FOR N = 1 TO R
740   IF A$(N)=N$ THEN A=N: GOTO 2000
750 NEXT N
760 IF A=0 PRINT "NAME NOT FOUND"
770 INPUT "DELETE ANOTHER NAME":E$
780 IF LEFT$(E$,1)="Y" GOTO 720
790 IF B=0 GOTO 5000
800 OPEN "O",1,D$
810 PRINT#1,R-B
820 FOR N= 1 TO R-B

830   PRINT#1,A$(N);",";B$(N)
840 NEXT N
850 CLOSE
860 GOTO 5000

2000 REM * DELETE & MOVE ALL UP ONE *
2010 FOR X = A TO R-B-1
2020   A$(X)=A$(X+1)
2030   B$(X)=B$(X+1)
2040 NEXT X
2050 A$(R-B+1)=" "
2060 B$(R-B+1)=" "
2070 B=B+1
2080 RETURN
```

R is the number of names on the file

Search list for name to be deleted

Test for another
If no deletions, there is no need to save file

Save new file under same name

Back to menu

Move names up one, erasing desired name and phone number

Blank out last name

Count number of deletions

SAVE THE NEW VERSION ON DISK AS DATFILE/BAS

Use of the New Section

Suppose we had a data file named TSTPH/TXT:0 on a disk and we had just input it from section 2 of DATFILE/BAS. The list is displayed.

```
ABLE B    555-Ø1Ø1
BAKER D   555-1Ø1Ø
CANDY K   555-1111
DUNKS C   333-2Ø2Ø
DUNKS D   333-2Ø2Ø
PRESS 'ENTER' TO CONTINUE?-
```

← This is the file called TSTPH/TXT:0

When you press ENTER, the menu is displayed.

```
PHONE DIRECTORY MENU

1. CREATE A NEW FILE
2. INPUT FILE FROM DISK
3. ADD RECORD(S)
4. DELETE RECORD(S)
5. EXAMINE RECORD(S)
6. QUIT

TYPE NUMBER OF SELECTION (1-6)?-
```

← You just used this

← This is next

Suppose you wish to delete the names BAKER D and CANDY K. Type in the number 4 and press the return key.

```
.
.
.
.
TYPE NUMBER OF SELECTION (1-6)?4
NAME TO BE DELETED?-
```

Type BAKER D

Exactly like it is in file

When the program finds BAKER A at line 740: N=2, A is set to 2, R=5, and B=0. Since A$(2) = N$ at line 740, the subroutine is entered.

During the execution of the FOR-NEXT loop:

FOR X = 2 TO 4		
		A$(1)=ABLE B
When X= A$(2)=A$(3)	→	A$(2) becomes CANDY K
B$(2)=B$(3)	→	B$(2) becomes 555-1111
When X=3 A$(3)=A$(4)	→	A$(3) becomes DUNKS C
B$(3)=B$(4)	→	B$(4) becomes 333-2020
When X=4 A$(4)=A$(5)	→	A$(4) becomes DUNKS D
B$(4)=B$(5)	→	B$(5) becomes 333-2020
Now out of FOR-NEXT loop		
A$(5)=""	→	A$(5) blanked out
B$(5)=""	→	B$(5) blanked out
B=B+1	→	B is increased to one (one change made)

The program returns to line 750. When it reaches line 770, a new question is posed.

```
.
.
.
NAME TO BE DELETED?BAKER D
DELETE ANOTHER NAME?-
NAME TO BE DELETED?-
```

Type YES

Type CANDY K

When the name, CANDY K, is found (now in A$(2)) at line 740; N=2, A is set to 2, R=5, and B=1. The subroutine:

```
FOR X = 2 TO 3
                                    A$(1) is still ABLE B
When X=2 A$(2)=A$(3)   ------>      A$(2) becomes DUNKS C
         B$(2)=B$(3)   ------>      B$(2) becomes 333-2020

When X=3 A$(3)=A$(4)   ------>      A$(3) becomes DUNKS D
         B$(3)=B$(4)   ------>      B$(3) becomes 333-2020

Now out of FOR-NEXT loop

         A$(4)="" "    ------>      A$(4) blanked out
         B$(4)="" "    ------>      B$(4) blanked out

         B=B+1         ------>      B is increased to 2
                                    (two changes made)
```

Once again a request is made to see if you want to delete another name. You answer NO and the program goes on to line 800 where the file is opened, the number of records is set to R-B (now 3), and the file is saved under the same name: TSTPH/TXT:0. The file is then closed and return is made to the menu.

```
PHONE DIRECTORY MENU

1. CREATE A NEW FILE
2. INPUT FILE FROM DISK
3. ADD RECORD(S)
4. DELETE RECORD(S)
5. EXAMINE RECORD(S)
6. QUIT

TYPE NUMBER OF SELECTION (1-6)?-
```

You might want to look at the altered file to make sure that the change was made correctly. If you do, type: 2 and press ENTER.

```
.
.
.
NAME OF FILE?TSTPH/TXT:Ø  <------ Type file name, press
                                  ENTER
```

```
ABLE B     555-Ø1Ø1
DUNKS C    333-2Ø2Ø   <------ Baker and Candy have
DUNKS D    333-2Ø2Ø          been removed and all
PRESS 'ENTER' TO CONTINUE?-  others moved up
```

Examine Records Section

One last section of the program remains. Suppose you would like to examine a data file by one of three different ways.

1. By selecting the left part of a person's last name. For example, C, CA, CAN, etc. would select CANDY K and D, or DU, DUN, etc. would select DUNKS C and DUNKS D.
2. By selecting a range of last names. For example, A to C would select ABLE B, BAKER D, and CANDY K and R to Z would select RIOT C and SAMPLE F. A to Z would select the complete file.
3. By selecting the phone prefix (first three digits). For example, 333 would select all those records with a 333 phone prefix and 555 would select all those records with a 555 phone prefix.

We'll show you one way to do it. You may be able to improve this section or add other ways to select records.

```
 9ØØ REM * EXAMINE RECORDS *
 91Ø CLS: PRINT "YOU MAY SEARCH FOR:"
 92Ø PRINT TAB(3)"1.ANY PART OF LAST NAME-FROM LEFT"
 93Ø PRINT TAB(3)"2.NAMES STARTING WITHIN A GIVEN RANGE"
 94Ø PRINT TAB(3)"3.BY FIRST 3 DIGITS OF PHONE #":PRINT
 95Ø PRINT "TYPE THE NUMBER OF YOUR SELECTION (1-3)"
 96Ø A$=INKEY$: IF A$="" GOTO 96Ø
 97Ø IF VAL(A$) <1 OR VAL(A$) >3 GOTO 95Ø
 98Ø ON VAL(A$) GOTO 21ØØ,22ØØ,23ØØ
 99Ø INPUT "WANT TO SEARCH SOME MORE";M$
1ØØØ IF LEFT$(M$,1)="Y" GOTO 91Ø
1Ø1Ø GOTO 5ØØØ

21ØØ REM * SEARCH FOR LAST NAME *
211Ø CLS: INPUT "LEFT LETTER(S) OF NAME":F$
212Ø L=LEN(F$): A=Ø
213Ø FOR N = 1 TO R
214Ø   IF LEFT$(A$(N),L)=F$ PRINT A$(N),B$(N): A=A+1
215Ø NEXT N
216Ø IF A=Ø PRINT "NONE FOUND"
217Ø GOTO 99Ø

22ØØ REM * SEARCH FOR RANGE *
221Ø CLS: INPUT "STARTING LETTER OF RANGE";F$
222Ø INPUT "ENDING LETTER OF RANGE";G$:A=Ø
223Ø FOR N = 1 TO R
224Ø   IF LEFT$(A$(N),1)>=F$ AND LEFT$(A$(N),1) <=G$ PRINT
       A$(N),B$(N):A=A+1
225Ø NEXT N
226Ø IF A=Ø PRINT "NONE FOUND"
227Ø GOTO 99Ø

23ØØ REM * SEARCH FOR PHONE PREFIX *
231Ø CLS: INPUT"FIRST 3 DIGITS OF PHONE #";P$:A=Ø
232Ø FOR N = 1 TO R
233Ø   IF LEFT$(B$(N),3)=P$ PRINT A$(N),B$(N): A=A+1
234Ø NEXT N
235Ø IF A=Ø PRINT "NONE FOUND"
236Ø GOTO 99Ø
```

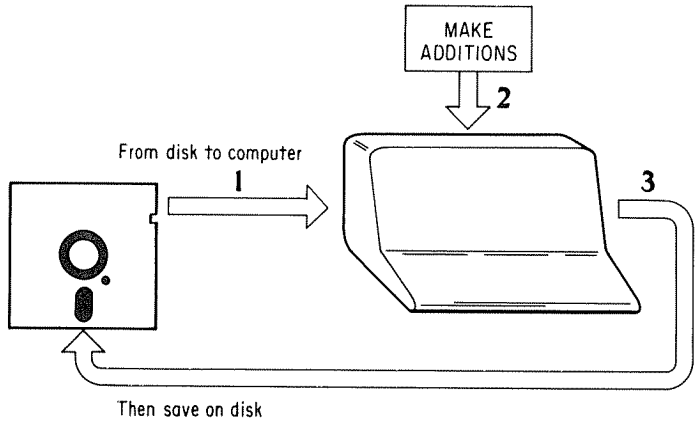
Input this section and SAVE on your disk the completed program DATFILE/BAS. You are now ready to use the complete directory.

Using the Completed DATFILE Program

Of course you must first access Disk BASIC and LOAD DATFILE/BAS before you can use it. After completing the housekeeping chores (lines 10 through 40), the menu is displayed.

```
PHONE DIRECTORY MENU

1. CREATE A NEW FILE
2. INPUT FILE FROM DISK
3. ADD RECORD(S)
4. DELETE RECORD(S)
5. EXAMINE RECORD(S)
6. QUIT

TYPE NUMBER OF SELECTION (1-6)?-
```

- You may create a new phone file by choosing selection 1. Any number of files may be created as long as each is given a distinct name.
- You may select any *previously created* file by choosing selection 2. Of course, the file must be on the disk that you are using.
- You may add records to an existing file (choice 3 from the menu) but again, you must first input that file from the disk (choice 2). This two-part selection is necessary because any operation performed on an existing file is done in the computer's memory. After the records have been added, the altered file must be saved on disk.



- You may delete records (choice 4), but again you must input the file to be altered (choice 2) from the disk. Then make the deletions and SAVE the altered file.
- You may examine records (choice 5). The file to be examined must also be input from the disk (choice 2) before it may be examined.
- Leaving the program is easy. Just type 6 when the menu is displayed. However, as a precaution, you should be sure that all files have been closed before removing a disk from the disk drive. When the computer says:

Type: CLOSE and press ENTER.

> The CLOSE command will CLOSE all files that have been in use. CLOSE all files before removing your disk from the disk drives. Then you may turn the computer off.

Try out selection 5 from the menu to see how it works.

1. Make sure DATFILE/BAS has been loaded.
2. Run DATFILE to get the menu.
3. LOAD PHONE/TXT; your data file.
4. Examine:
   a) records starting with:
      1) AB
      2) DU
      3) FI
      4) COB
   b) records whose names range from:
      1) C–G
      2) A–H
      3) I–Z
      4) D–A
   c) records whose phone prefixes are:
      1) 555
      2) 333
      3) 444

Here are the results of our check of section 5:

1. Load DATFILE

```
HOW MANY FILES?
MEMORY SIZE?
RADIO SHACK DISK BASIC VERSION 2.2
READY
>LOAD "DATFILE/BAS"
```
← LOAD it

2. Run DATFILE/BAS

```
HOW MANY FILES?
•
•
•
>LOAD "DATFILE/BAS"
READY
>RUN
```
← Run it

3. Load PHONE/TXT

```
        PHONE DIRECTORY MENU

        1. CREATE A NEW FILE
        2. INPUT FILE FROM DISK
        3. ADD RECORD(S)
        4. DELETE RECORD(S)
        5. EXAMINE RECORD(S)
        6. QUIT

        TYPE NUMBER OF SELECTION (1-6)?2
NAME OF FILE?PHONE/TXT
```
← You type

Computer ——→ lists names and phone numbers
You ——→ press ENTER
Computer ——→ prints directory again

4. Examine records

Type 5 in answer to the selection number.

```
•
•
•
     TYPE NUMBER OF SELECTION (1-6)?5
 YOU MAY SEARCH FOR:
    1. ANY PART OF LAST NAME-FROM THE LEFT
    2. NAMES STARTING WITHIN A GIVEN RANGE
    3. BY FIRST 3 DIGITS OF PHONE #
 TYPE THE NUMBER OF YOUR SELECTION (1-3)?1
```
← for check 4a)

4a)

4a1)

```
LEFT LETTERS OF NAME?AB
ABLE B    555-Ø1Ø1
WANT TO SEARCH SOME MORE?YES
YOU MAY SEARCH FOR:
   1. ANY PART OF LAST NAME-FROM THE LEFT
   2. NAMES STARTING WITHIN A GIVEN RANGE
   3. BY FIRST 3 DIGITS OF PHONE #
TYPE THE NUMBER OF YOUR SELECTION (1-3)?1
```

One name beginning with AB

4a2)

```
LEFT LETTERS OF NAME?DU
DUNKS C    333-2Ø2Ø
DUNKS D    333-2Ø2Ø
WANT TO SEARCH SOME MORE?YES
   •
   •
TYPE THE NUMBER OF YOUR SELECTION (1-3)?1
```

Two names beginning with DU

4a3)

```
LEFT LETTERS OF NAME?FI
FINK F    333-1212
WANT TO SEARCH SOME MORE?YES
   •
   •
TYPE THE NUMBER OF YOUR SELECTION (1-3)?$
```

One name beginning with FI

4a4)

```
LEFT LETTERS OF NAME?COB
NONE FOUND
WANT TO SEARCH SOME MORE?YES
YOU MAY SEARCH FOR:
   1. ANY PART OF LAST NAME-FROM THE LEFT
   2. NAMES STARTING WITHIN A GIVEN RANGE
   3. BY FIRST 3 DIGITS OF PHONE #
```

No names beginning with COB

4b)

```
TYPE THE NUMBER OF YOUR SELECTION (1-3)?2
```

Go on to next selection

4b1)

```
STARTING LETTER OF RANGE?C
ENDING LETTER OF RANGE?G
CANDY K    555-1111
DUNKS C    333-2Ø2Ø
DUNKS D    333-2Ø2Ø
EVERS L    555-213Ø
FINK F     333-1212
GOOD K     555-312Ø
WANT TO SEARCH SOME MORE?YES
   •
   •
   •
TYPE THE NUMBER OF YOUR SELECTION (1-3)?2
```

From C through G

4b2)

```
STARTING LETTER OF RANGE?A
ENDING LETTER OF RANGE?H
ABLE B     555-Ø1Ø1
BAKER D    555-1Ø1Ø
CANDY K    555-1111
DUNKS C    333-2Ø2Ø
DUNKS D    333-2Ø2Ø
EVERS L    555-213Ø
FINK F     333-1212
GOOD K     555-312Ø
HENRY B    555-2212
WANT TO SEARCH SOME MORE?YES
  •
  •
  •
TYPE THE NUMBER OF YOUR SELECTION (1-3)?2
```

From A through H

4b3)

```
STARTING LETTER OF RANGE?I
ENDING LETTER OF RANGE?Z
IRVING D    333-1115
JONES E     555-7632
KANE K      555-76Ø3
LEMON R     333-1515
MARKS H     555-8119
NOBLE M     333-8991
OTTO D      333-1129
PINKS C     555-321Ø
RIOT C      555-4567
SAMPLE F    333-7654
WANT TO SEARCH SOME MORE?YES
  •
  •
  •
TYPE THE NUMBER OF YOUR SELECTION (1-3)?2
```

From I through Z

4b4)

```
STARTING LETTER OF RANGE?D
ENDING LETTER OF RANGE?A
NONE FOUND
WANT TO SEARCH SOME MORE?YES
  •
  •
  •
```

Can you search backward?

No

4c)

```
TYPE THE NUMBER OF YOUR SELECTION (1-3)?3
```

Next type of selection.

4c1)

```
FIRST 3 DIGITS OF PHONE #?555
ABLE B     555-Ø1Ø1
BAKER D    555-1Ø1Ø
CANDY K    555-1111
EVERS L    555-213Ø
GOOD K     555-312Ø
HENRY B    555-2212
JONES E    555-7632
KANE K     555-76Ø3
MARKS H    555-8119
PINKS C    555-321Ø
RIOT C     555-4567
WANT TO SEARCH SOME MORE?YES
  •
  •
  •
TYPE THE NUMBER OF YOUR SELECTION (1-3)?3
```

All names whose phone prefix is 555

4c2)

```
FIRST 3 DIGITS OF PHONE#?333
DUNKS C    333-2Ø2Ø
DUNKS D    333-2Ø2Ø
FINK F     333-1212
IRVING     333-1115
LEMON P    333-1515
NOBLE M    333-8991
OTTO D     333-1129
SAMPLE F   333-7654
WANT TO SEARCH SOME MORE?YES
  •
  •
  •
TYPE THE NUMBER OF YOUR SELECTION (1-3)?3
```

All names whose phones begin with 333

4c3)

```
FIRST 3 DIGITS OF PHONE#?444
NONE FOUND
WANT TO SEARCH SOME MORE?NO
```

Any with a 444 prefix?
NO

All done

The computer goes back to the menu.

```
        PHONE DIRECTORY MENU

        1. CREATE A NEW FILE
        2. INPUT FILE FROM DISK
        3. ADD RECORD(S)
        4. DELETE RECORD(S)
        5. EXAMINE RECORD(S)
        6. QUIT

        TYPE NUMBER OF SELECTION (1-6)?6
READY
>CLOSE
READY
```

Let's quit

Make sure all files are closed

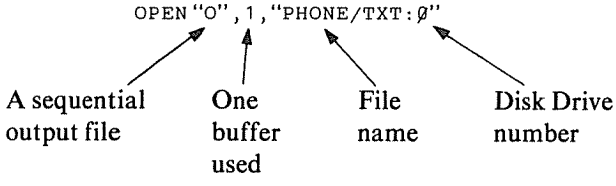
Summary

A program was developed in this chapter to demonstrate the use of a disk data file using names and telephone numbers. It can serve as a model, or as a starting point, for you to create disk files of your own. The program was broken into the following functional modules that were selectable from the keyboard.

a) create a new file from the keyboard,
b) input a previously created file from disk,
c) add one or more records to the file,
d) delete one or more records from the file, and
e) examine one or more records in the file.

You also learned:

- to OPEN a file,



If no disk Drive number is specified, Drive zero (0) will be used.

- to CLOSE a file,

CLOSE ← closes all files that have been opened

CLOSE 1 ← closes the file assigned to buffer #1

CLOSE 1, 2, 8 ← closes files assigned to buffers #1, 2, and 8

- all open files should be CLOSEd before removing a disk from the disk drive,
- to use the DATA FILE program to:
  a) create a disk data file consisting of names and phone numbers and to save it on disk,
  b) input a previously created file from disk to the computer's memory,
  c) add one or more records to the file,
  d) delete a record from the file and move all records following it up one position,
  e) examine one or more records by:
     1. last name or the left part of the last name
     2. all last names starting within a specified range
     3. telephone prefix

- PRINT #1 is used to transfer data from the computer's memory to disk,
- INPUT #1 is used to transfer data from a disk file to the computer's memory, and
- LINE INPUT is used to input strings containing commas, quotes, and other punctuation marks.

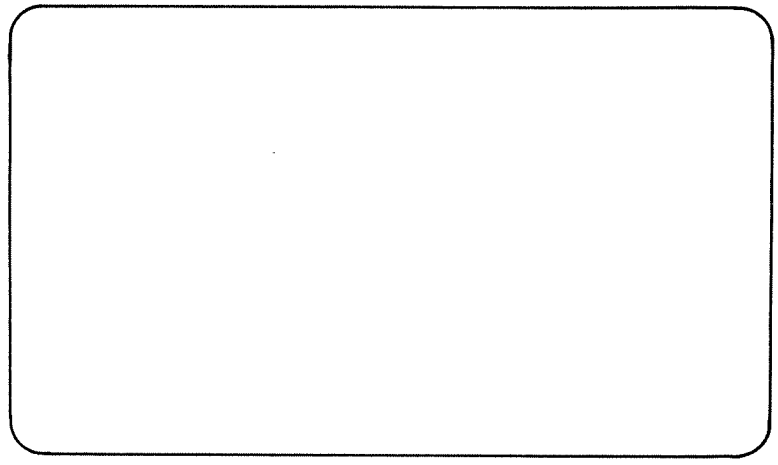
Self-Test

1. Use the completed DATFILE/BAS program to create a file from the keyboard containing the following names and phone numbers.

CABLE A	353-1111
FORCE S	535-8055
GROSS B	353-7546
KING E	556-3294
MORE L	636-4288

Save the file on disk as MYLIST/TXT.

2. Use DATFILE/BAS to input the data list saved in exercise 1. Copy what you see on the screen after the file is input.



Then make the following changes:

Delete: KING E
CABLE A

Add:

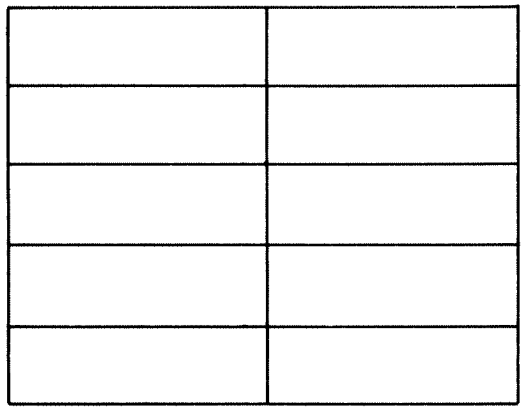
SOZO W	335-7876
PARKS C	335-0132
WIZARD Z	353-2109

3. Use DATFILE/BAS to input the modified file of exercise 2. Examine the file and list the resulting names and phone numbers.

Name	Phone

4. Use the existing DATFILE/BAS program to arrange the names in alphabetical order. DO NOT USE THE CREATE-A-NEW-FILE SECTION.

5. Input the new MYLIST/TXT file and copy the file here.



6. Eliminate "MYLIST/TXT" from your disk and create a personal phone file for your own use.

Answers to Self-Test

1. Access Disk BASIC; load and run "DATFILE/BAS"; select #1 from the menu and enter the five names and phone numbers. Save as MYLIST/TXT.

2. Select menu selection #2.

```
NAME OF FILE?-
```
← MYLIST/TXT

```
CABLE A    353-1111
FORCE S    535-8Ø55
GROSS B    353-7546
KING E     556-3294
MORE L     636-4288
PRESS 'ENTER' TO CONTINUE?-
```

Select #4 and delete KING E and CABLE A.
Select #2 and input the modified file. You see:

```
FORCE S    535-8Ø55
GROSS B    353-7546
MORE L     636-4288
```

Select #3 and add SOZO, PARKS, and WIZARD.

3. Select #2 from the menu.

```
FORCE S    535-8Ø55
GROSS B    353-7546
MORE L     636-7546
SOZO W     335-9876
PARKS C    335-Ø132
WIZARD Z   353-21Ø9
```

4. Select #4 and delete PARKS, SOZO, and WIZARD.
   Select #2 and read back.

```
FORCE S    535-8Ø55
GROSS B    353-7546
MORE L     636-4288
```

Select #3 and add in order PARKS, SOZO, and WIZARD.

5. Select #2 from the menu.

```
NAME OF FILE?-
```

MYLIST/TXT

```
FORCE S    535-8Ø55
GROSS B    353-7546
MORE L     636-4288
PARKS C    335-Ø132
SOZO W     335-9876
WIZARD Z   353-21Ø9
```

6. Select #6.

```
READY
>-
```

Type: `KILL "MYLIST/TXT"`

Then work on your own personal file.

CHAPTER EIGHT

Tuning Up Your Computer

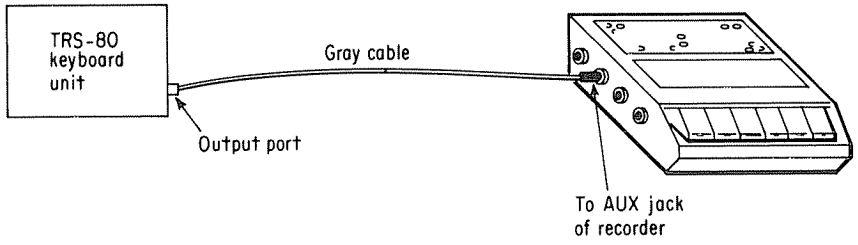
You already know how to put graphics on the video display to add action to your BASIC programs. Wouldn't it be nice to add sound to your programs to really liven things up? In this chapter you will learn:

- how to connect your computer to an inexpensive speaker/amplifier to produce sounds,
- that the cassette connections of your computer can be used to feed signals to the speaker/amplifier,
- how to control pulses to the speaker/amplifier with the OUT 255,2 statement to turn on a pulse and the OUT 255,0 statement to turn off a pulse,
- how electrical pulses are converted to sound waves,
- how to use a machine language subroutine from your BASIC program to produce sounds,
- how to save memory space for the machine language subroutine by setting MEMORY SIZE?,
- how to access the machine language subroutine with the basic statement: X = USR(0), and
- how the Radio Shack program, MICRO MUSIC, is used.

Graphics, discussed in chapter 3, bring live action to the computer's video screen. The generation of sound to accompany your programs adds another dimension to your computer.

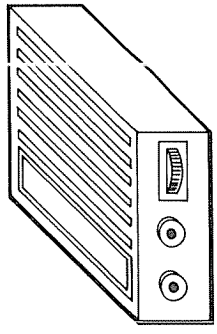
You can develop your own hardware interface and software driver to produce sound. You can also buy commercially available hardware and software if you wish. However, a combination of these two possibilities is more fun. The necessary equipment is easy to obtain and use.

You have probably used the TRS-80 cassette recorder to CSAVE and CLOAD programs. If you read the earlier chapters of this book on cassette files (and we hope you have tried all demonstration programs and exercises), you know that the cassette recorder can save and retrieve data files. Let's consider how the computer saves information on tape. Data is sent from the computer as a series of electrical pulses to an output port. The output port is connected by means of a gray cable to the AUX jack of the recorder.

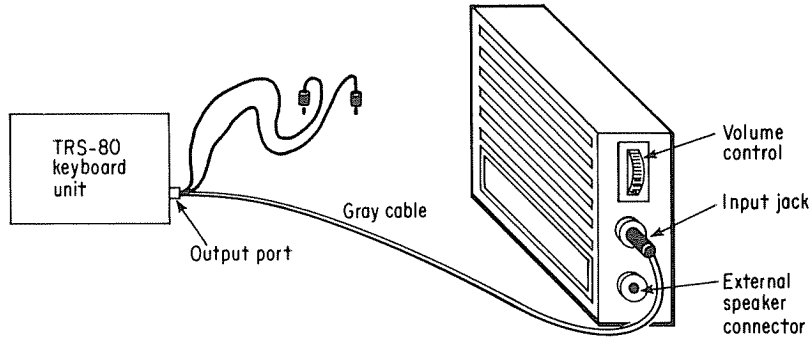


Ordinarily, these pulses are recorded on the magnetic tape of the cassette so that they can be CLOADed back into the computer at a later time.

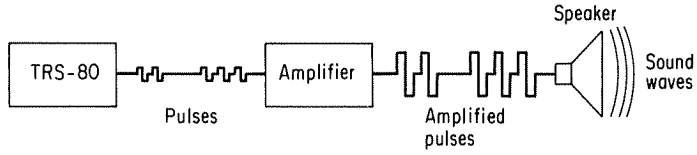
If you disconnect the cassette cable (the black one) that connects the computer and the EAR jack of the recorder and play a tape recording that has been CSAVEd, you will discover that you can hear the recorded pulses. Thus, if you could control the pulses sent out by the computer, you might be able to produce sounds of your own. If you want to immediately hear the sounds that you produce, you can send the pulses to an amplifier/speaker combination instead of to the recorder. Radio Shack stores sell a combination speaker/amplifier unit for only $11.95.



If you connect the gray cable (which ordinarily goes to the AUX jack of the recorder) to the input jack of the speaker/amplifier, you will have a sound system that is sufficient for your needs.



The pulses from the computer are amplified so that they can drive (make the speaker vibrate) the speaker hard enough to produce sound waves.



You now have a hardware system that can produce sounds. You must find a way (a program) to create the electrical pulses.

The cassette's output port (the place where the three cassette cables leave the computer) has been assigned a number (255) that the computer knows. A signal is sent to the cassette's output port by sending the number 2 to the port. The signal is turned off by sending a zero (0) to the cassette port.

```
OUT 255,2
```
Output — To port 255 — A 2 — Turns on a signal

```
OUT 255,Ø
```
Output — To port 255 — A zero — Turns off the signal

A combination of these two statements will create the following pulse:

Turn on your computer, hook up the speaker/amplifier as shown on page 213, and enter this BASIC program.

Sound Generator

```
11Ø FOR X = 1 TO 5Ø
12Ø   OUT 255,2   <-- Pulse
13Ø   OUT 255,Ø   <--
14Ø   FOR Z = 1 TO 2Ø: NEXT Z   <-- Time delay
15Ø NEXT X
```

Run the program and you will hear the result of fifty pulses created by the FOR-NEXT loop (X as variable). A time delay controls the frequency of the sound. The longer the delay (FOR Z = 1 TO ?), the lower the frequency will be. By changing line 140, the frequency will be changed.

FOR Z= 1 TO 10: NEXT Z
High frequency

FOR Z = 1 TO 20: NEXT Z
Low frequency

The outside FOR-NEXT loop (variable X) creates the duration of the sound. The higher its upper value, the longer the sound is heard.

FOR X = 1 TO 10

Long duration

FOR X = 1 TO 3

Short duration

To demonstrate the possibilities for changing the frequency of the sound, change line 140 to:

```
14Ø FOR Z = 1 TO F: NEXT Z
```

The frequency can then be varied by adding a FOR-NEXT loop (lines 100 and 160).

```
1ØØ FOR F = 19 TO 1 STEP -2

16Ø NEXT F
```

List the program. It should now look like this:

Sound Generator

```
LIST

1ØØ FOR F = 19 TO 1 STEP -2
11Ø   FOR X = 1 TO 5Ø
12Ø     OUT 255,2
13Ø     OUT 255,Ø
14Ø     FOR Z = 1 TO F: NEXT Z
15Ø   NEXT X
16Ø NEXT F
```

Duration 50

Frequency values: 19, 17,15,13,11, 9,7,5,3,1

Delay between pulses controlled by frequency values

Run the program and you will hear the note rise in tone as the frequency value (F) goes down from 19 to 1. Although a change in tone can be detected, the results are not very satisfactory.

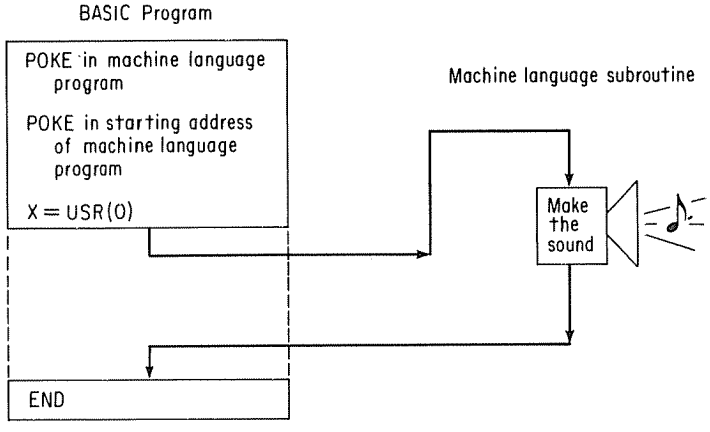
The time necessary to interpret the BASIC statements is too long for the creation of realistic sounds. The speaker must vibrate faster than the BASIC statements are executed. To get the necessary speed, we will have to write a subroutine in the computer's own language so that it doesn't have to take time translating what we want done. You can switch from a BASIC language program to a machine language subroutine by using a new BASIC function (USR).

The User Function

The user function is our link between our BASIC program and the machine language subroutine that will produce the sounds. We will use the BASIC program to POKE the machine language subroutine into memory. To access the machine language subroutine, we'll use the statement:

```
X = USR(Ø)
```

When the computer executes the USR statement, it immediately looks into memory locations 16526 and 16527 to see where the machine language subroutine can be found (the memory location where it begins).



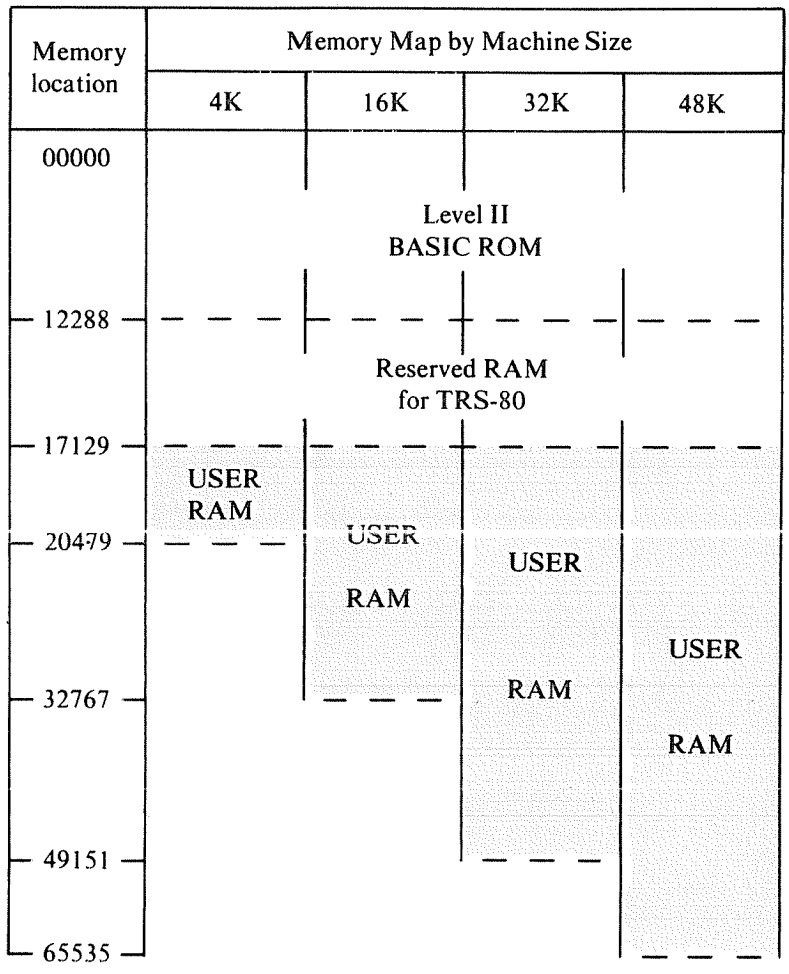
Saving Memory for the Machine Language Subroutine

Remember the first message that appears on the video screen when the computer is turned on?

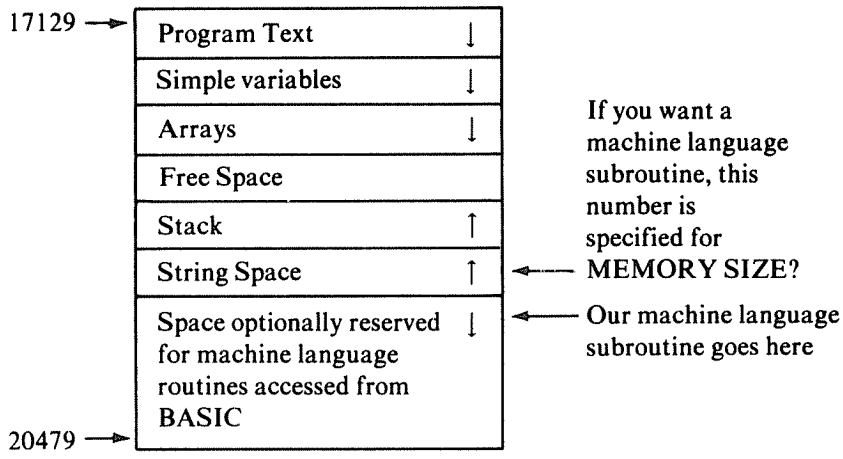
```
MEMORY SIZE?
```

Now is your chance to answer with a number instead of merely pressing the ENTER key. In order to know what number to enter at this point, refer to the memory maps that follow.

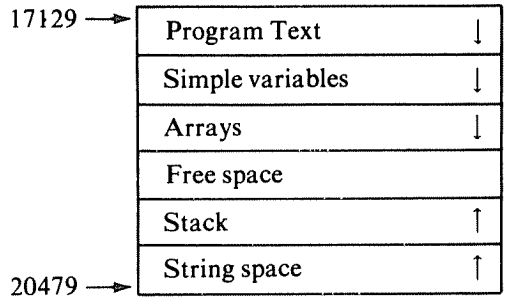
The first map shows the area (shaded) that may be used by you for your programs. Let's stay within the restrictions of a 4K TRS-80 (memory locations 17129 through 20479). In that way our program will be machine-independent (it won't matter whether you are using a 4K, 16K, 32K, etc.).



In the following memory map, the user space for a 4K TRS-80 is shown.



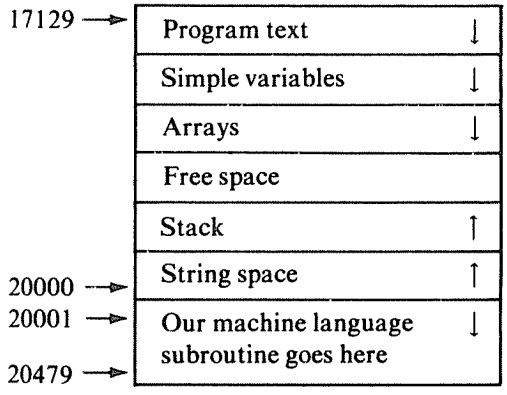
When you press the ENTER key without specifying a number for MEMORY SIZE, the optional space at the top of memory is *not* reserved. String space would be used from 20479 downward.



If you do specify a memory address following the MEMORY SIZE prompt, space *will* be reserved for a machine language subroutine.

Example:

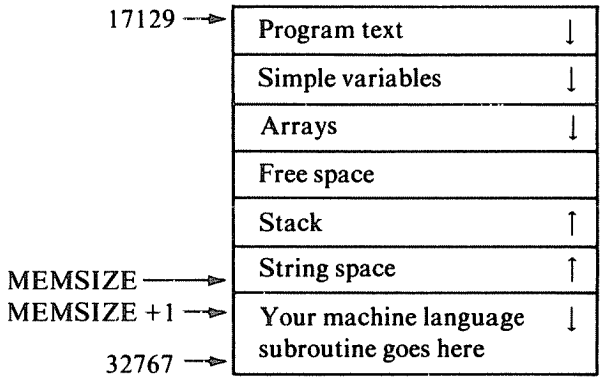
```
MEMORY SIZE?20000
```



Memory locations 20001 through 20479 will be reserved for a machine language subroutine and the string space will be allocated from 20000 downward. The memory location specified at MEMORY SIZE time is *one less than* the location where your machine language subroutine will begin.

For a 16K machine, the only difference would be the upper memory boundary (32767 instead of 20479). However, if you are using a 16K machine, you will probably want to set MEMORY SIZE to a higher location (such as 32000). Every location above the MEMORY SIZE response is saved for machine language.

For a 16K TRS-80 the memory map looks like this:



Now let's see how you can produce some sounds by storing a machine language subroutine in the reserved space.

The BASIC Program

The first section of the program clears the screen so that there will be no distractions when the sounds are produced. It then calls a subroutine which POKEs (see chapter 2 for a description of the POKE statement) the machine language program into memory.

Sound Producing Program
Section 1

```
100 REM ** CLEAR SCREEN - POKE MACHINE LANGUAGE **
110 CLS: GOSUB 1000
```

The second section plays eight notes of a musical scale by means of the machine language subroutine, which is called by the USR function at line 240. The frequency of the notes is read from a data list and POKEd into the subroutine to replace the note previously played.

Section 2

```
200 REM ** CHANGE NOTE AND PLAY **
210 R=255: POKE 20002,R: L=0: POKE 20004,L
220 FOR Y = 1 TO 8
230   READ F: POKE 20020,F
240   X=USR(0)
250   FOR W = 1 TO 100: NEXT W
260 NEXT Y
270 END
```

The third section contains the data for the machine language subroutine in lines 310 and 320. These values are the decimal equivalents of the machine language instruction codes. Remember they are actually put into memory in binary format. The data for the frequency of the notes is contained in line 330.

Section 3

```
300 REM ** SUBROUTINE AND FREQUENCY VALUES **
310 DATA 14,16,33,0,1,58,61,64,230,253,198,2,211,255,214
320 DATA 2,211,255,6,160,16,254,43,124,181,32,239,201
330 DATA 85,80,75,70,65,60,55,50
```

All that is left is the subroutine that is called from Section 1 to read in the data values given in Section 3.

POKE Subroutine

```
1000 REM ** DATA READING SUBROUTINE **
1010 POKE 16527,78: POKE 16526,33
1020 FOR X = 20001 TO 10028
1030   READ D
1040   POKE X,D
1050 NEXT X
1060 RETURN
```

Now you're ready to enter the program. Turn off your TRS-80 if it is on. Wait a few seconds, then turn it back on again. When it prompts you with:

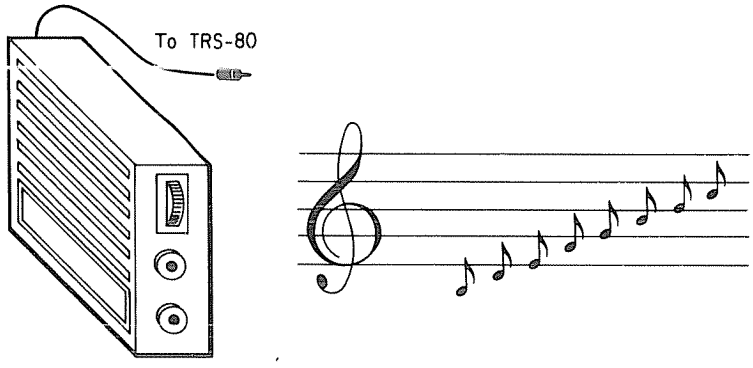
```
MEMORY SIZE?-
```

answer by typing in the value 20000.

```
MEMORY SIZE? 20000
RADIO SHACK LEVEL II BASIC
READY
>-
```

Enter the program. However, before you RUN it, make sure that the DATA values in lines 310 through 330 are correct. A mistake in lines 310 or 320 could cause a real disaster, even if only one item is wrong. Line 330 is just the data for the notes and is not as critical as the other two data lines.

Once you have double checked the data statements, RUN the program, and the scale is played. Be sure the speaker/amplifier is connected, turned on, and the volume control turned up high enough to hear the notes.



By experimenting with different values for the variables R, L, and F in the Sound Producing Program, you can discover different types of sounds. If you find some appropriate sounds you can incorporate them into other BASIC programs.

Adding Sound To Your Programs

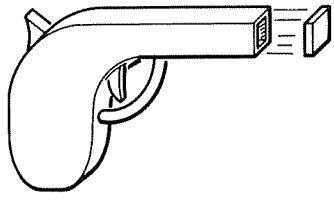
After experimenting with the notes and noises that can be produced with your new sound system, we have come up with some samples to go with our demonstration programs.

For the first demonstration, we'll use the program from chapter 3 that painted the screen white and punched black holes in it. We have added the sound of a bullet "puncturing" the screen. Here is the program with the sound modifications.

Target Practice Program

```
100 REM ** POKE SOUND **
110 CLS: GOSUB 1000

200 REM ** PAINT THE SCREEN **
210 CLS
220 FOR S = 0 TO 1022
230   PRINT @S,CHR$(191);
240 NEXT S

300 REM ** FIRE 100 SHOTS **
310 FOR W = 1 TO 100
320   S=RND(1023)-1: PRINT @S,CHR$(128):
330   X=USR(0)
340   FOR Z = 1 TO 150: NEXT Z
350 NEXT W

400 REM ** DO IT AGAIN **
410 PRINT @900,"PRESS 'ENTER' TO CONTINUE": INPUT A$
420 GOTO 210

1000 REM ** POKE SOUND SUBROUTINE **
1010 POKE 16527,125: POKE 16526,1
1020 FOR Y = 32001 TO 32030
1030   READ D: POKE Y,D
1040 NEXT Y
1050 RETURN
1060 DATA 14,1,6,80,58,61,64,230,253,198,2,211,255,214,2,211
1070 DATA 255,197,16,254,193,16,242,13,121,246,0,32,234,201
```

To run this program, the MEMORY SIZE? prompt must be set to 32000 when the TRS-80 is turned on. Imagine that you have a gun with a rectangular barrel that



shoots rectangular shaped bullets. RUN the program and the gun shoots rectangular holes through the white screen with the accompanying sound as the screen is punctured. The screen is painted white, then:

POW! A black hole is shot in the screen
POW! Another black hole
POW! Another

This goes on until one hundred shots have been taken. There may not be one hundred holes in the screen since some shots may go through a hole that is already there.

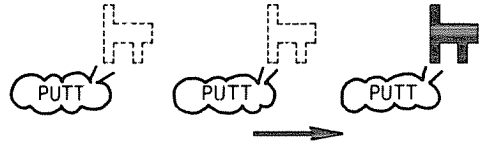
You may wish to change some of the data to vary the sound that is produced. A value that you might try changing is the 80 (fourth item in line 1060).

A second demonstration uses the "running" of a single car across the screen (See chapter 3). We'll add some sound to the program. The sound subroutine of the previous program is used again, but a change is made in the fourth data item, which controls the pitch of the noise produced.

Car with Sound Program

```
 100 REM ** POKE SOUND **
 110 CLS: GOSUB 1000
 200 REM ** MAKE SOUND AND MOVE CAR **
 210 FOR M = 15360 TO 15420
 220   X = USR(0)
 230   POKE M+3,156: POKE M+2,140
 240   POKE M+1,170: POKE M,128
 250 NEXT M
 260 GOTO 260

1000 REM ** POKE SOUND SUBROUTINE **
1010 POKE 16527,125: POKE 16526,1
1020 FOR Y = 32001 TO 32030
1030   READ D: POKE Y,D
1040 NEXT Y
1050 RETURN

1100 DATA 14,1,6,40,58,61,64,230,253,198,2,211,255,214,2,211
1110 DATA 255,197,16,254,193,16,242,13,121,246,0,32,234,201
```

MEMORY SIZE is again set to 32000. The subroutine produces a short tone each time the car moves one rectangle to the right.



Variations in the sound subroutine could produce different noises as the car moves across the screen. Our "putt-putt" sound is not the best. Try our own variation by changing the fourth data item in line 1100.

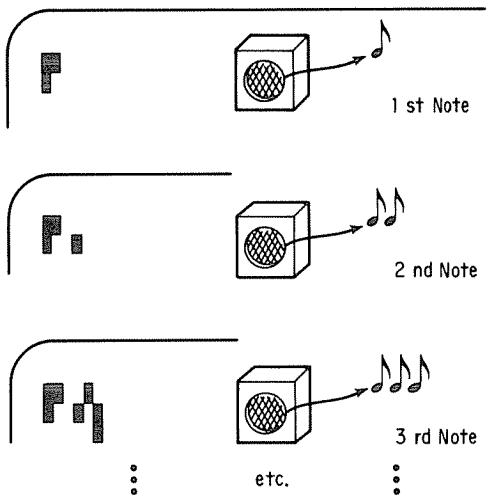
Now we return to the Mandala Program introduced in chapter 3. This time, we'll produce a tone after each of the four symmetrical graphic symbols are displayed.

Mandala with Sound Program

```
100 REM ** SET UP MACHINE LANGUAGE **
110 CLS: GOSUB 1000
120 S = 10: Z = 250: POKE 32020,Z

200 REM ** DRAW MANDALA **
210 FOR N = 0 TO 330 STEP 66
220   FOR R = 0 TO S
230     A=RND(2)-1:B=RND(2)-1:C=RND(-1):D=RND(2)-1:E=RND(2)-1:
240     F=RND(2)-1
250     IF N=0 THEN M=0 ELSE M=62*(N/66)
260     UL=128+A+2*B+(2*C)↑2+(2*D)↑3+(2*E)↑4+(2*F)↑5
270     UR=128+B+2*A+(2*D)↑2+(2*C)↑3+(2*F)↑4+(2*E)↑5
280     LL=128+E+2*F+(2*C)↑2+(2*D)↑3+(2*A)↑4+(2*B)↑5
290     LR=128+F+2*E+(2*D)↑2+(2*C)↑3+(2*B)↑4+(2*A)↑5
300     PRINT @544,CHR$(191);:PRINT @144+N+R,CHR$(UL);
310     PRINT @176+M-R,CHR$(UR);:PRINT@912-M+R,CHR$(LL);
320     PRINT @944-N-R,CHR$(LR);
330     X = USR(0)
340   NEXT R
350   Z = Z-5: POKE 32020,Z
360   S = S-1
370 NEXT N

400 REM ** WAIT AWHILE **
410 FOR WAIT = 1 TO 500: NEXT WAIT
420 RESTORE: CLS: GOTO 120

1000 REM ** SOUND SUBROUTINE **
1010 POKE 16527,125: POKE 16526,1
1020 FOR Y = 32001 TO 32026
1030   READ D: POKE Y,D
1040 NEXT Y
1050 RETURN

1100 REM ** MACHINE LANGUAGE DATA **
1110 DATA 14,255,33,0,255,58,61,64,230,253,198,2,211
1120 DATA 255,214,2,211,255,6,250,16,254,37,32,241,201
```

MEMORY SIZE is set to 32000. After drawing four symmetrical graphic symbols, a tone is played. The tone variable (Z) is lowered (this raises the tone) for the display of each line of graphic characters.

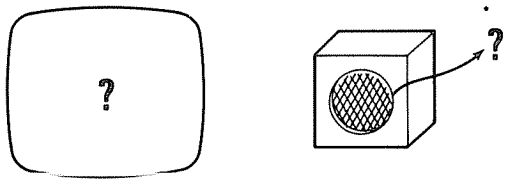
Example: (Only upper left corner of screen shown)



For a variation, try switching lines 330 and 340.

```
33Ø NEXT R
34Ø X = USR(Ø)
```

How will this change the program?



Now a single note will play after each line of the mandala is displayed, instead of after each four symbols. The note rises in pitch each time.

Our last sound demonstration program makes use of several BASIC statements that you can incorporate into other programs. It allows you to fill the screen with text. A warning sound is included for the end of each line. A little birdie "chirps" when you are within five characters of the end of a line. A different sound is included that sounds after each keystroke is made. A third sound is given each time you press the ENTER key after each line of text is completed.

After you have filled the screen with text, the display is cleared and the entire message is again displayed. A fourth sound provides a surprise ending.

The sounds are produced in one of four subroutines. The text is stored in a string array labeled B$(N) that is dimensioned for 16 strings (one for each line of the video screen). Each string in the array is built up in a loop that allows up to 64 characters in the string. This loop can be exited by pressing the ENTER key to terminate that particular string. The INKEY$ function is used to enter each character that is added to the current string being formed.

Write Text Program

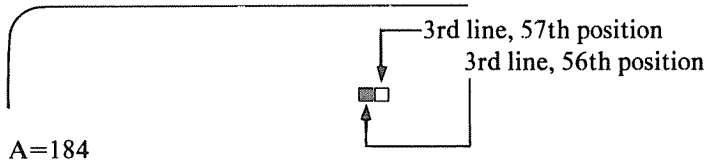
```
 1Ø REM *INITIALIZE*
 2Ø CLEAR 15ØØ                                  <- Save string space
 3Ø POKE 16527,125: POKE 16526,1: GOSUB 1ØØØØ
 4Ø DIM B$(16)                                  <- 16 lines of strings
 5Ø CLS

1ØØ REM *FILL THE SCREEN WITH TEXT*
11Ø FOR N=1 TO 16                               <- Loop for 16 lines
12Ø   C = 1                                     <- 64 characters per line;
13Ø   A=PEEK(16416)+PEEK(16417)*256               Get cursor position
14Ø   IF (A/64-INT(A/64))*64=57 GOSUB 1ØØØ
15Ø   A$=INKEY$: IF A$="" THEN 15Ø ELSE PRINT A$;
16Ø   IF ASC(A$)=8 THEN C=C-1: B$(N)=LEFT$(B$(N),C-1):GOTO 13Ø
17Ø   IF A$=CHR$(13) THEN GOSUB 2ØØØ; GOTO 22Ø  <- Check for ENTER key
18Ø   GOSUB 3ØØØ
19Ø   B$(N)=B$(N)+A$                            <- Add A$ to string
2ØØ   C = C+1: IF AC<63 THEN GOTO 13Ø
21Ø   GOSUB 2ØØØ
22Ø NEXT N


23Ø REM * CLEAR SCREEN AND REPRINT TEXT*
24Ø FOR W=1 TO 5Ø: NEXT W
25Ø CLS
26Ø FOR W=1 TO 2ØØ: NEXT W
27Ø FOR N= 1 TO 15
28Ø   PRINT B$(N)
29Ø   GOSUB 2ØØØ
3ØØ NEXT N
31Ø PRINT B$(N);
32Ø GOSUB 4ØØØ
33Ø GOTO 33Ø

1ØØØ REM ** RING BELL **
1Ø1Ø POKE 32ØØ4,2: POKE 32Ø2Ø,15Ø
1Ø2Ø X = USR(Ø); FOR W = 1 TO 2Ø: NEXT W
1Ø3Ø X = USR(Ø)
1Ø4Ø RETURN

2ØØØ REM ** CARRIAGE RETURN SOUND **
2Ø1Ø FOR Y = 1 TO 5
2Ø2Ø   POKE 32Ø2Ø,255
2Ø3Ø   X = USR(Ø)
2Ø4Ø NEXT Y
2Ø5Ø RETURN

3ØØØ REM ** KEY CLICK **
3Ø1Ø POKE 32Ø2Ø,2
3Ø2Ø X = USR(Ø)
3Ø3Ø RETURN
```

```
4000 REM ** SURPRISE ENDING **
4010 FOR Y = 50 TO 100 STEP 5
4020    POKE 32020,Y
4030    X = USR(0)
4040    FOR W = 1 TO 50; NEXT W
4050 NEXT Y
4060 RETURN

10000 REM **POKE MACHINE LANGUAGE **
10010 FOR Y = 32001 TO 32026
10020    READ D: POKE Y,D
10030 NEXT Y
10040 RETURN
10050 DATA 14,255,33,0,20,58,61,64,230,253,198,2,211
10060 DATA 255,214,2,211,255,6,150,16,254,37,32,241,201
```

In line 130, the variable A is assigned the current cursor position on the video screen. This value is held in memory locations 16416 and 16417. The value changes with each keystroke. Since the decimal number 255 is the largest value that will fit in any one memory location, the cursor's position on the screen (a memory location between 15360 and 16383, inclusive) must be broken into two parts. Memory location 16416 holds the least significant part of the cursor's position. Memory location 16417 holds the most significant part of the cursor's position. This later value must, therefore, be multiplied by 256 before it is added to the least significant part, to form the complete decimal value of the cursor position. Line 140 compares the current position to the number 57 to see if it is equal to that position on the given line. If it is, a GOSUB statement causes the warning bell at subroutine 1000 to ring.

Examples:

a) Cursor at position 184



A=184
A/64 = 2.875
INT(A/64) = 2
(A/64–INT(A/64))*64=56 Don't ring bell yet

b) Cursor at position 185

A=185
A/64 = 2.890625
INT(A/64) = 2
(A/64–INT(A/64))*64=57 GOSUB 1000 and ring bell

After a key is pressed, the character is printed by line 150. If the ENTER key was pressed, line 170 calls the carriage return sound subroutine at line 2000. On return from the subroutine, the GOTO statement (end of line 170) causes the next line of text to be started. If some other key is pressed, line 180 calls the key click subroutine at line 3000. The character is then added to (concatenated) the string at line 190.

The last section of the main program clears the screen and then prints the string array, giving the carriage-return sound at the end of each line. A bonus sound is provided to indicate the end of the text. Press the BREAK key and type RUN if you want to type in a new screen of text.

The demonstration that follows is fairly self-explanatory. The end-of-line warning bird chirps when there are five or less spaces left on the line you are typing. Any line can be terminated by pressing the ENTER key (as on line 1). Empty lines can be executed by pressing the ENTER key more than once. Key clicks are produced after each keystroke. A surprise bonus sound is produced when the last line of the text is reprinted on the screen.

Here is a demonstration using the program:

```
THIS IS A DEMONSTRATION OF THE WRITE TEXT PROGRAM. <-- Press ENTER
AFTER PRESSING RETURN, TYPE THE SECOND LINE. THE BELL RINGS.
PRESS ENTER TO GET TO THIS THIRD LINE. THIS TIME WE WILL TYPE U <-- Automatic return
NTIL THE CARRIAGE IS AUTOMATICALLY RETURNED. NEXT TIME SPACE
     IN 5 SPACES BEFORE TYPING ADDITIONAL TEXT. PRESS RETURN
WHEN YOU HEAR THE BELL OR FINISH THE WORD IF THERE IS ENOUGH
ROOM. YOU CAN ALSO TERMINATE THE LINE AT ANY TIME
BY PRESSING THE ENTER KEY AS WE JUST DID. YOU CAN ALSO HEAR
THE KEY CLICKS AS EACH LETTER IS TYPED. PRESS ENTER TWICE NOW

AND YOU WILL SKIP ONE LINE. WITH A LITTLE MORE WORK, YOU COULD
MAKE THIS PROGRAM INTO A SIMPLE WORD PROCESSOR.


THE ENTER KEY WAS PRESSED THREE TIMES AFTER THAT LAST LINE.
TWO LINES WERE SKIPPED. NOW LISTEN AND WATCH AS YOU FINISH.
```

MICRO MUSIC

Another commercial, sound-producing program is called MICRO MUSIC and is available from Radio Shack stores for around $14.95. It comes in cassette form with both Level I and Level II versions on the same tape. MICRO MUSIC, as the title suggests, is used to produce music. It lets you type a tune directly on the video screen using a letter for each individual note (C,D,E,F,G,A, or B).

You can hear the results of your composition in several ways. You can connect the plug that usually goes to the recorder to a Hi-Fi system. You may also use the inexpensive Radio Shack amplifier/speaker discussed earlier in this chapter. Another method is to record the results on tape using the cassette recorder in the normal way. Then you can play back the tape to hear the music.

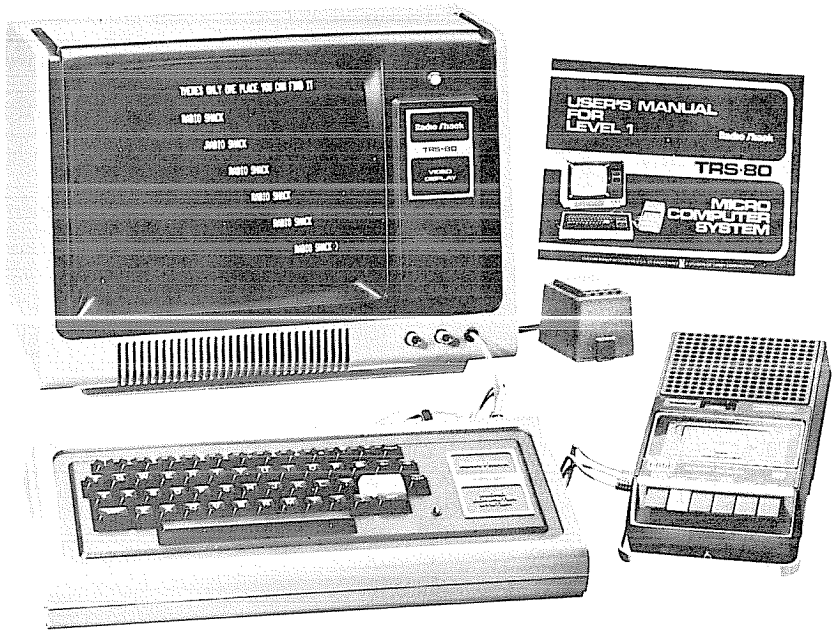
MICRO MUSIC has a five-octave range including normal notes, sharps, and flats. Three different tone qualities can be programmed into your music. You can program whole notes, half notes, quarter notes, eighth notes, dotted notes, and triplets. You also have a choice of two basic tempos. You can repeat sections of your music up to nine times and even use an alternate ending the last time the section is played.

Each note and its duration are displayed at the bottom of the video screen as it is played. Your tape recorder is automatically started and stopped each time you want to record your music. Editing features allow modification of your song between plays, making it easy to get your composition just the way you want it. Instructions for using MICRO MUSIC are provided with the cassette software. These instructions include:

a) general description,
b) load and use instructions,
c) editing features,
d) instructions on how to write your own music (with several short examples),
e) a discussion of advanced features,
f) instructions on saving and loading the composition, and
g) a reference sheet of functions.

Using MICRO MUSIC

Let's look at some features of the MICRO MUSIC software as it is used on the following Radio Shack system.



In addition, the Radio Shack amplifier/speaker shown on page 158 is used.

On a Level II TRS-80, the software is loaded in the SYSTEM mode since it is written in machine language. When the system is turned on, you see:

```
MEMORY SIZE-
```

press the ENTER key.

```
MEMORY SIZE?
RADIO SHACK LEVEL II BASIC
READY
>_
```

Type: SYSTEM and press ENTER, then wait until the asterisk (*) appears on the screen.

```
MEMORY SIZE?
RADIO SHACK LEVEL II BASIC
READY
>SYSTEM
*
```

Type: MUSIC and press ENTER, then wait for the second asterisk.

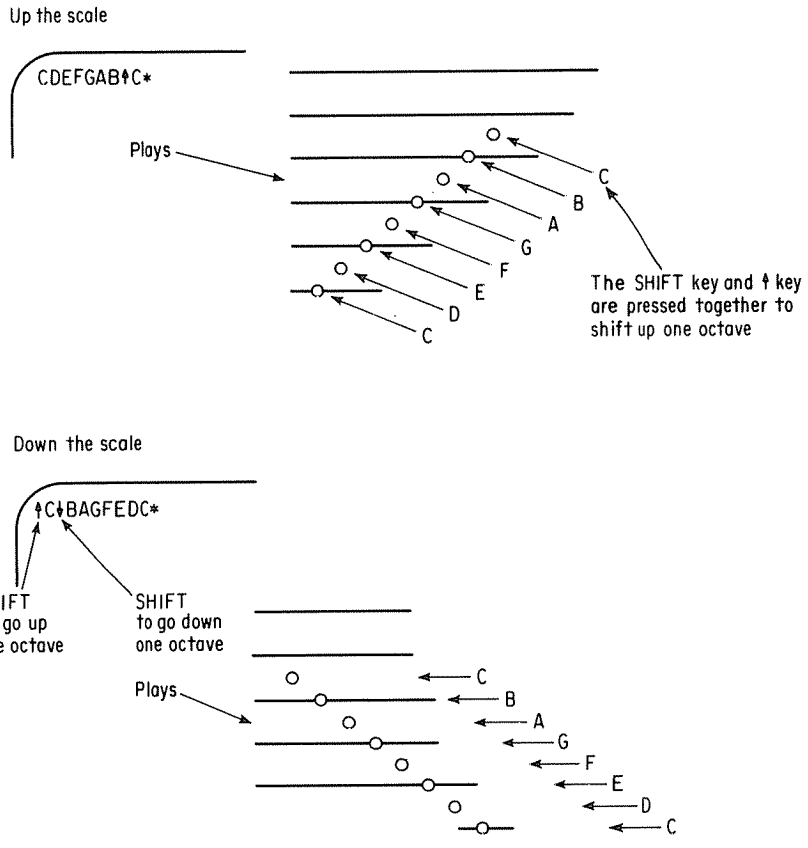
```
MEMORY SIZE?
RADIO SHACK LEVEL II BASIC
READY
>SYSTEM
*MUSIC
*
```

Type: a backslash (/) and press ENTER, then wait until a flashing asterisk appears in the upper left corner of the screen.

```
*
```

Now you can type in the notes to be played.

Examples:



A Typical Amateur's Composition

Music can be composed directly from sheet music using MICRO MUSIC. Even if you've never played a musical instrument, you can be composing your own music in no time at all. If it doesn't sound right, you can easily change your arrangement with the editing features.

Composer's own version of Scarborough Fair

```
(3D2D4A4G4A4E4.F8E4D2.A2↑C4D8C8D8D8.C8↓A4B4G4A2A4↑D2 ↓D4F2G4A4G4F
4D8C8C4D4D2A4G2F4E4D4C4D2R4D2D4A4G4E4.F8E4D2.A2↑C4D8C8D8D4.C8↓A4
B4G4A2A4↑D2↓D4F2G4A4G4F4D8C8C4D4D2A4G2F4E4D4C4D2R4)RRR
```

Special Features

The following special features are used by MICRO MUSIC.

Keyboard Entry	Function Resulting
B#	play B sharp – whole note
B–	play B flat – whole note
B2	play B – half note
B4	play B – quarter note
B8	play B – eighth note
B. or B2. or B4. or B8.	play B 1½ times the stated length
SHIFT ↑	shift to next higher octave
SHIFT ↓	shift to next lower octave
L	change tone quality (to thinnest)
M	change to double time
N	back to normal from triplet, stacatto, or tone
R	rest
S	stacatto
T	speed up for triplets
V	change tone quality (thinner than normal)
W	slow to half speed
Y	play in high range (upper 3 octaves)
Z	play in bass range (lower 3 octaves)

Edit Functions

a) Move the cursor left, right, up, or down.
b) Clear screen for new music.
c) Change to command mode.
d) Begin playing music.
e) Insert a blank space.
f) Delete a character.
g) Interrupt the music.

Summary

In this chapter, we have shown one method to produce sounds on the TRS-80.

- We made use of the cassette output port and a Radio Shack Speaker/Amplifier to make the sounds.
- We found that BASIC was not fast enough to make useful sounds. Therefore, we resorted to writing a machine language subroutine that was accessed through a BASIC program by means of the USR function.
- To enter the machine language subroutine, we had to reserve a block of memory for it. This was done by inputting a memory location when the computer was turned on in response to the prompt:

```
MEMORY SIZE?
```

- The POKE statement was used to place the machine language subroutine in memory when the BASIC program was run.
- The machine language program was executed by the BASIC statement:

```
X = USR(Ø)
```

- Several demonstrations were given to show how sound can be added to previous BASIC programs.
- The Radio Shack program, MICRO MUSIC, was discussed and directions on its use were given.

Self-Test

1. Signals are sent from the TRS-80 Computer to the cassette recorder's (REMOTE, EAR, AUX) jack.
2. When we used our simple sound system, we connected the cable which normally goes to the recorder to send the sounds to a _____
3. BASIC language proved to be too (slow, fast) to produce useful sounds.
4. A machine language subroutine can be accessed from a BASIC program by means of the _____ function.
5. How do you save memory space for a machine language program?

   _____

   _____
6. Your machine language subroutine should begin at the memory location that is (one more than, one less than, the same as) the number specified for MEMORY SIZE.
7. MICRO MUSIC, a cassette program to produce your own music, is available from _____
8. The software for MICRO MUSIC is written in (BASIC language, machine language)
9. To use MICRO MUSIC, you type in _____ and _____ that represent the notes to be played.
10. MICRO MUSIC can play whole, _____, _____, _____ and _____ notes.

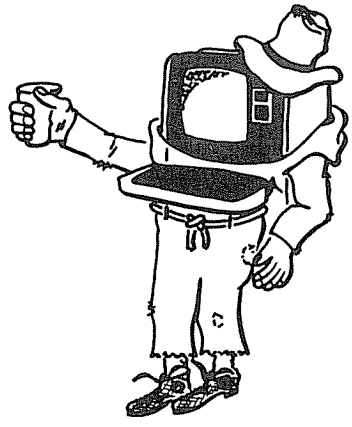
Answers to Self-Test

1. AUX
2. Speaker/Amplifier
3. slow
4. USR
5. Input a memory location in response to the computer's MEMORY SIZE? prompt
6. one more than
7. Radio Shack stores
8. machine language
9. letters and numbers
10. half, quarter, eighth, and dotted

CHAPTER NINE

Special Features and Fancy Functions

In chapter 2 you learned to save memory space by squeezing several BASIC statements into a single program line. Of course, doing so makes a program difficult to read, but when you need the space, every extra byte is appreciated.



This chapter shows some other ways to save memory in your TRS-80 and introduces the powerful error handling capabilities of your small computer. Let's start by doing things with the BASIC language that use as little memory as possible.

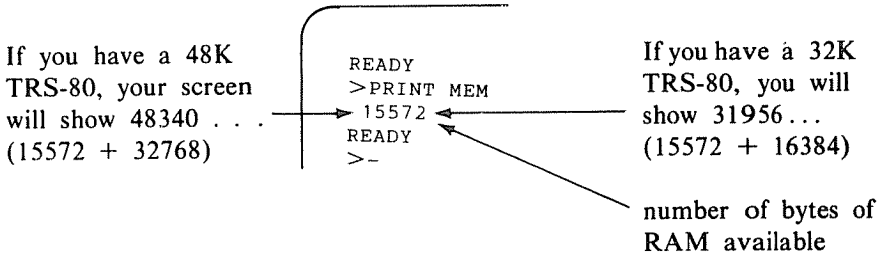
Take Small Bytes First

We have an older Level II 16K TRS-80. If you have a newer machine or one with more memory, most of the following demonstrations will produce individual results that differ with what shows on our screen. Don't worry about that. Go ahead and try the examples. The memory savings (the differences between any two results) discussed are the same for all machines, regardless of computer memory size.

To begin, type NEW and then ask the TRS-80 to give you the number of bytes of free memory space by typing:

```
PRINT MEM
```

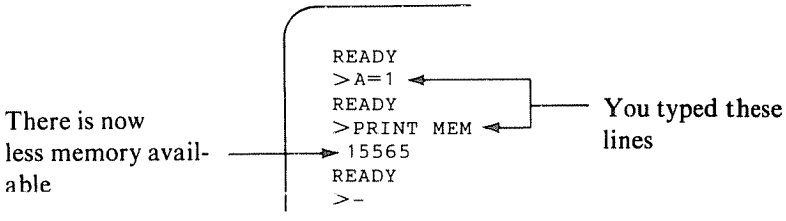
ReMEMber! MEM is a TRS-80 *reserved* word for the *variable* name that contains the number of bytes of RAM available.



Now try this example. Type: NEW. With the screen cleared, type:

```
A=1
PRINT MEM
```

Your screen should show the following information:



Before setting the variable A to one, 15572 bytes were available. When the variable A was set to one (A=1), the MEMory count went down to 15565, 7 fewer bytes of free memory. *(For those of you with larger or newer machines:* The number on your screens will also be down by 7 bytes. Machines with 32K will show 31949; 48K will show 48333. No matter what your screen shows, just watch the *differences,* and you can follow along with the discussion.)

Seven bytes! The TRS-80 used up 7 bytes to reserve a place in memory for the variable A (the place where the number 1 is now stored) and to record the name of the variable. In fact (we will tell you how we know this later) the TRS-80 used 4 bytes to store the number and 3 bytes to store the name. Four plus three *is* seven. There must be a way to save some of this space. There is. . . . Read on.

!! IMPORTANT MESSAGE !!
DO NOT SKIP

In the last example and in those that follow, if you make a typing mistake that results in a ?SYNTAX ERROR, start the example over from the *beginning*. Start by typing the word NEW and then the rest of the example.

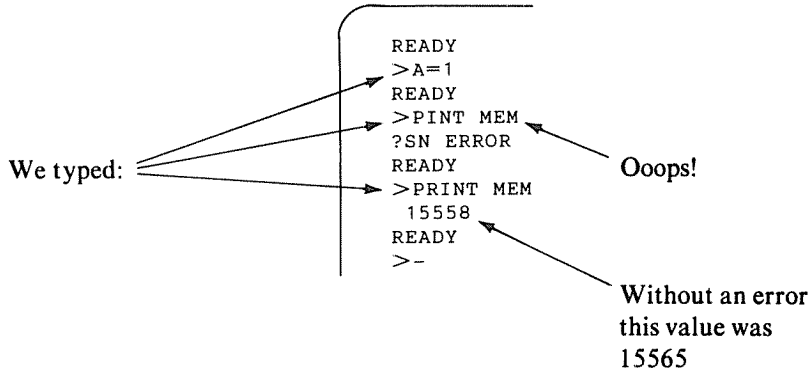
Why? Because typing an error can cause the TRS-80 to use memory space. This extra use of memory throws off the memory count in MEM, and the results on the screen will be other than what is shown in the book.

Here is a deliberate entry of an error to demonstrate what occurs. We type NEW to clear the screen. Then we type:

```
A=1
PINT MEM
PRINT MEM
```

Ooops!!

The screen shows:



So, if an error occurs while you are entering an example in this section, *stop* and reenter the example from the beginning.

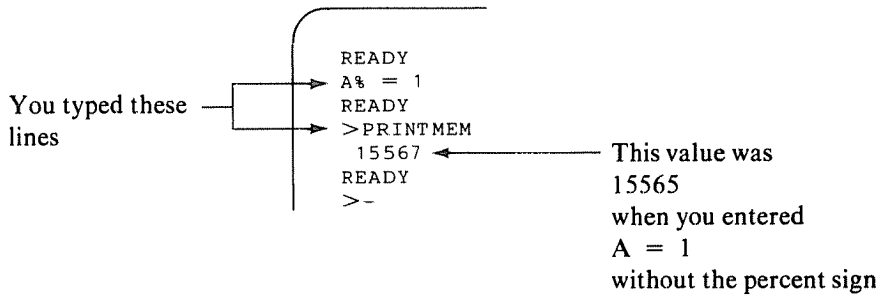
!! IMPORTANT MESSAGE !!

If you have not read the message shown above, please do so now. O.K., how can you save some memory space? Enter the next example, and observe what happens to the value of MEM.

Type the following into your TRS-80:

```
NEW
A%=1
PRINT MEM
```

Your screen should show the following information:



By placing the percent sign after the variable name, two bytes of memory were freed for use. Two bytes doesn't seem like much at this point, but wait. A byte here and a byte there soon adds up to a lot of extra memory. Best of all, you can have the TRS-80 take care of the bookkeeping problems of how to make these savings.

The percent sign after the variable name told the TRS-80 to treat A% as an *integer*. That is, A% would not need a decimal point and would never have any fractional values (values requiring numbers after the decimal point). Knowing this, the TRS-80 can conserve on the amount of memory needed for this variable. When the variable A is used without the percent sign, the TRS-80 reserves seven locations in memory (three for the name; four for the value). For A%, five locations are reserved (three for the name; two for the value).

A Quick Look at Precision

For most numeric variables, such as those in earlier examples of (A=1), the TRS-80 reserves 7 bytes of memory. Three bytes store the variable's name and four bytes store the variable's value. Numbers stored in this form are called *single precision* numbers and the variables are called *single precision* variables. A later chapter discusses in detail exactly how a number is represented in the 4 bytes and how it is retrieved and used in arithmetic. For now, learn the terminology so you can see how much space is used when you put numbers into the TRS-80. The goal in this chapter is to talk about the use of space.

You have already seen that variables with a percent sign (%) after the variable name take less memory to store numbers. *Integer* variables, as these type of variables are called, use only 5 bytes of memory. Here is a short table that compares some features of single precision and integer variables:

Type of Variable	Bytes used			Value ranges	
	Name	Value	Total	Smallest	Largest
Integer	3	2	5	−32768	+32767
Single precision	3	4	7	−1.701411E+38	+1.70411E+38

As you can see, integer variables take up less memory but can only represent numbers that are much smaller than single precision variables. Unless you put a percent sign after a variable name, the TRS-80 automatically makes the variable single precision. The TRS-80 also allows you to *explicitly* tell it that a variable is single precision by putting an exclamation point after the variable name. For example:

```
A! = 1
```

Tells the TRS-80 explicitly to make the variable single precision

produces the same result as using the assignment statement A=1.

If you enter one of the earlier examples but use an exclamation point (!) this time, you can show that the TRS-80 does work this way.

Enter the following statements:

```
NEW
A! = 1
PRINT MEM
```

Your screen should display:

```
READY
>A! = 1
READY
>PRINT MEM
 15565
READY
>-
```

The same use of memory as when you typed A=1

So, A=1 and A!=1 produce the same result in terms of the amount of memory being used. More on what single precision is about later. For now, continue to look at how to save some space.

Take a Few More Bytes

Space saving or space usage begins to become a significant factor when you use arrays.

Enter the following into your TRS-80:

```
NEW
A!(Ø) = 1
PRINT MEM
```

Assigns the value "one" to the single precision (!) array A! — first element position A!(0)

Type in the lines. Your screen should indicate this result:

```
READY
>A!(Ø) = 1
READY
>PRINT MEM
 15520
READY
>-
```

Hmm . . .
This example used
a lot of memory

The available memory locations have been reduced by 52 bytes!! Whew! The assignment of *one* number into the array A! caused the TRS-80 to use up 52 bytes.

$$(15572 - 15520 = 52)$$

Try one more experiment before you attempt to puzzle out what has just occurred. Enter the following lines into the computer:

```
NEW
A%(Ø) = 1
PRINT MEM
```

The percent sign tells the
TRS-80 to set
up an integer array

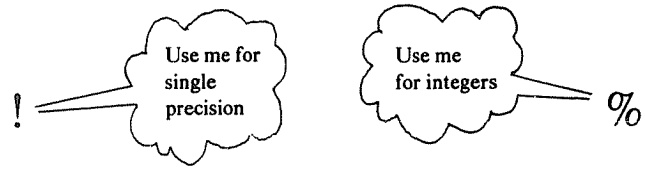
Does your screen show this display?

```
READY
>A%(Ø) = 1
READY
>PRINT MEM
 15542
READY
>-
```

Down 30 bytes from
15572

Are you beginning to see any patterns in these results? Well, the difference between the number of bytes used for the A! array and the A% array is 52 minus 30, or *22 bytes*. Aha! Each array contains 11 elements (since no DIMension statement was used). Two times 11 *is* 22 — the number of bytes difference in the two values of memory used. You saw previously that an integer variable required two less bytes of memory. For this set of experiments, each array had 11 elements so. . . . You get the idea?

Arrays of integer values also take up two less bytes per element of the array. As the number of array elements gets larger, the amount of memory space you can save by using integer arrays becomes proportionally larger.



Two Thousand Bytes Is Worth a . . .

Using single precision or integers for large arrays can have significant effects on the amount of memory you use in a program.

Enter and RUN the short program that follows:

1000 elements ←

```
100 CLS
110 DIM A!(999)
120 PRINT MEM
```

When you ran the program did the number 11536 appear at the top of your screen? If not, make certain you typed the program *exactly* as shown above. Put in the spaces shown in lines 110 and 120. When your program looks like the one above, RUN it again.

O.K., now change line 110, as follows:

```
110 DIM A%(999)
```
← 1000 integer elements

RUN this program. Does your screen now show 13536? Yes! Good! Let's see, 13536 minus 11536 is 2000 — *2000* bytes of memory difference between the two small programs! All you did was change the ! to a % — a change from single precision to integer variable names. *The motto is:* Use integer arrays, if you can and save yourself a lot of memory.

Answer to the Unasked Question

Some of you (those of you who like puzzles) are probably still a few pages back trying to unravel the mystery of how the TRS-80 came up with 52 and 30 bytes, respectively, for the use of the unDIMensioned A! and A% arrays. Without going into a lot of detail, here is a table that shows the TRS-80 memory allocations for arrays of these types:

	Bytes Used				
Type of Array Variable	Array Name	Array Size	Number of Dimensions	Each Dimension	Each Element
Integer (!)	3	2	1	2	2
Single Precision (%)	3	2	1	2	4

Information in the table shows that the unDIMensioned integer array A% requires the following number of bytes:

- 3 for the name
- 2 for the size of the array
- 1 for the number of dimensions
- 2 for the single dimension it is using
- 22 for the number of elements (2 times 11)

The total is 30 bytes. For a single precision array the count is the same except for the last item in the list. The array A! would require 4 times 11, or 44 bytes, for its element storage. Adding this number of bytes to the eight required for the name, size, number of dimensions, and the single dimension yields 52 bytes. The mystery is solved, for now. You will soon have a few more to puzzle over.

The TRS-80 "Bytes" Back

Enter the following sequence of commands into your TRS-80:

```
NEW
A! = 1 + 2/3
PRINT A!
A% = A!
```

Put A! into A%

Here is what your screen should show:

```
READY
>A! = 1 + 2/3
READY
>PRINT A!
 1.66667
READY
>A% = A!
READY
>-
```

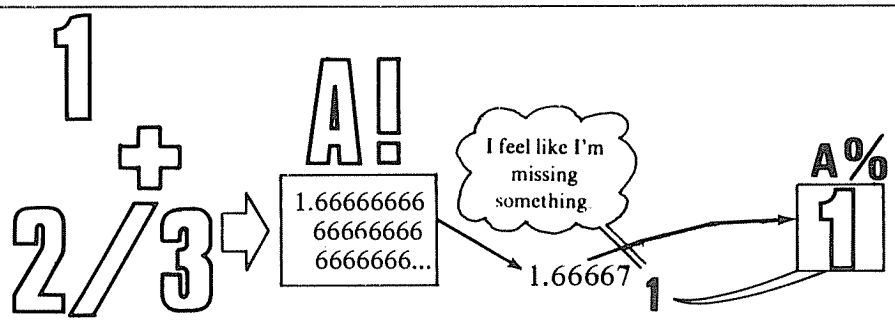
Yes, 1 plus 2/3 is 1.666666 . . . that rounds out to 1.66667

What will this do?

Can you guess what the screen displays when this statement is entered?

```
PRINT A%
```

If you said the number one (1), you're correct. When A% is set equal to A!, only the integer portion of the number is stored into A%. The fractional portion of A! is *not* carried over. So, *beware!* If you are using integer and single precision numbers together, you can experience inadvertent truncation of results in the computer.



Double Trouble

Some applications, mostly scientific and engineering problems, require *more* precision for the numbers and calculations used. If you want to know why, look up some friendly neighborhood scientists and have them explain their reasons for this need. As for the TRS-80, it is ready to handle those problems. Try the following examples and see how the TRS-80 accomplishes this feat.

Many science applications use the constant *pi* in calculations. *Pi* is the ratio of the distance around a circle to the circle diameter. The formula for this relationship is:

$$\text{circumference} = pi \times \text{diameter}$$

So, *pi* is equal to the circumference (distance around the circle) divided by the diameter (distance across the circle). *Pi* (often represented by the Greek symbol π) turns out to be a constant for *all* circles. The value of *pi or* π is:

$$pi = 3.141592653589\ldots$$

The *ellipsis* (three dots at the end) indicates that the number goes on forever. *Pi* can be represented by as many digits as you want to use, but you always have to truncate some portion of the never-ending stream of numbers it takes to represent *pi exactly.* (It is not possible to actually show it exactly, since the numbers go on forever and never repeat.) Things become complex when you begin to deal with the world of science!

But, suppose you need *pi* to be represented in the TRS-80 with at least as much accuracy as shown above. Try it and see what happens.

Enter the number and have the TRS-80 PRINT it on the screen for you:

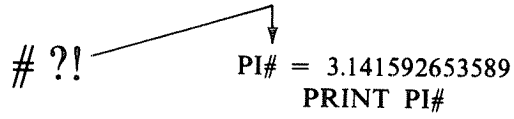
```
PI = 3.141592653589
PRINT PI
```

Does the result look like this?

```
>PI = 3.141592653589
READY
>PRINT PI
 3.14159 <-------- Where's the rest?
READY
>-
```

The TRS-80 truncated the last seven digits of the number. Hmm . . . let's see. *Pi* is a *single* precision variable. What is needed is a variable with more precision — something like a *double* precision variable.

Try this:



Anything surprising show up on your screen? Yes? What? Did your TRS-80 remember all of the digits this time?

```
>PI# = 3.141592653589
READY
>PRINT PI#
 3.141592653589 <-------- Yep! There's
READY                     the rest.
>-
```

Placing a # symbol after a variable name tells the TRS-80 to reserve space for a *double precision* number. But, what does this cost in terms of the number of bytes of memory?

Comparing Bytes

If you run the same set of comparisons done earlier on the amount of memory used by double precision variables versus integer and single precision, you would discover that double precision numbers require 11 bytes of memory. That amount of memory is 4 bytes larger than a single precision number. If it still takes only 3 bytes to store the name, then a double precision variable uses 8 bytes for the storage of the value. Yes, that is *double* the number of bytes used for value storage by a single precision variable. The following table summarizes these results:

Type of Variable	Bytes Used: Name	Value	Total		
Integer	3	2	5	%	Use these symbols behind the variable name
Single Precision	3	4	7	!	
Double Precision	3	8	11	#	

You also get comparable results for double precision arrays.

Enter the following:

```
NEW
A#(Ø) = 1
PRINT MEM
```

Your screen should show:

```
READY
>A#(Ø) = 1
READY
>PRINT MEM
 15476 <-------- That's 96 bytes used
READY
>-
```

For arrays, double precision variables use 4 bytes more per array element than single precision arrays, and 6 bytes more per element than integer arrays. Here is a brief summary of the space used by unDIMensioned arrays and arrays containing 1000 elements:

Array Name	Number of Bytes Used: UnDIMensioned (11 elements)	DIMensioned (1000 elements)	
A%	30	2008	<-- Integer
A!	52	4008	<-- Single Precision
A#	96	8008	<-- Double Precision

Each array requires 8 bytes to store the name, size of the array, number of dimensions, and space for each dimension (one dimension in this case). The rest of the space is used to store the values to be put into the array.

Double Warning! Double Warning!

Take care when letting the TRS-80 convert single precision numbers to double precision.

Enter the following and observe the results:

```
NEW
A# = 1/3
PRINT A#
```

Look at the screen carefully:

```
READY
>A# = 1/3
READY
>PRINT A#
 .3333333432674408          Strange!!
READY
>-
```

The fraction one-third is another repeating number that can be represented by a l-o-n-g string of threes after a decimal point.

1/3 = .33333... Forever →

The TRS-80 screen does not show this result. The last nine digits of the displayed number are "garbage." Only the first seven digits are accurate. How can this be fixed? Simple! Just enter the constant into A# this way:

```
A# = 1/3#
```
← Put symbol at end

When you PRINT A# after making this assignment, the screen shows:

.33333333333333333 17 digits

String Space

So far, all of this chapter's discussions have dealt with numeric variables and arrays. What about string variables and string arrays? How do they use memory space?

Begin your exploration of string space by entering the following into your TRS-80:

```
NEW
A$ = "1"
PRINT MEM
```

Good! What do you see? Does it look like this?

```
READY
>A$ = "1"
READY
>PRINT MEM
 15566
READY
>-
```
← 6 bytes used

That operation used 6 bytes. If it also takes 3 bytes for the name of the variable (like all the rest you have looked at), then the other 3 bytes must be used to store the string. Right? Well . . . let's see.

Try this set of entries:

```
NEW
A$ = "123456789"
PRINT MEM
```
9 characters

Huh?! Did you get the same number of bytes of MEMory available?

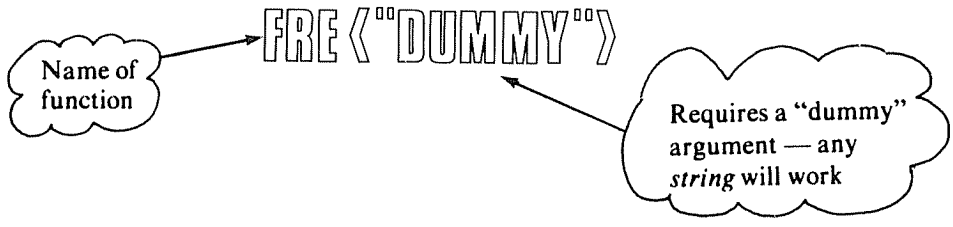
```
READY
>A$ = "123456789"
READY
>PRINT MEM
 15566
READY
>-
```

Same number!

How can this be?

Oh, yes! The TRS-80 puts strings in a special block of memory called the *string space.* When you first turn the machine on, 50 bytes of memory are reserved for string operations. (Review chapter 2 for more discussions of this feature.) You can reserve more or less space by using the CLEAR command. CLEAR 0 releases the string space for other use; CLEAR 1000 would increase the string space to 1000 bytes.

The difficulty is that when the characters of a string are placed into the string space, the value of MEM does not change. As more strings go into the space, the space fills, and, if the space is exceeded, an error message occurs indicating that you have run out of string space. You need a way to tell how much string space is being used to avoid having the program stop with an error. The TRS-80 lets you do this for FREe. (Oops, we gave it away.)



Type the following statement into the TRS-80, and observe the result produced:

```
PRINT FRE("D")
```

Any string will work here

The screen should show:

```
>PRINT FRE("D")
 41
READY
>-
```

The number of bytes of *FREE* memory in the string space

The available string space is 50 minus 9, or 41 bytes. Nine bytes were used in the last example when you entered A$="123456789". There are nine characters in the string and that takes up nine memory locations. So, string variables behave somewhat like numeric variables, except the total amount of memory used depends on the size of the string being used. The same is true for string arrays. This table summarize these results:

	Bytes Used		
	Single Variable	Arrays	
Array Name	(Includes 3 bytes for name)	unDIMensioned (11 elements)	DIMensioned (1000 elements)
A%	5	30	2008
A!	7	52	4008
A#	11	96	8008
*A$	6	41	3008

*Note: A$ also uses 1 byte for *every* character that appears in every string. The total space used is a function of the size of the strings stored in A$.

Naming Names and Saving Space

By now, many of you have probably realized that using the characters %, !, #, and $ to indicate variable types in a program can lead to a lot of errors in typing. Also, the extra characters use memory space themselves in simply storing the lines of the program. The solution: TRS-80 special BASIC language features called *type definitions.*

The TRS-80 allows you to specify and control how variables and constants are handled in the TRS-80 memory with statements that *DEF*ine variables to be of a particular type (integer, single precision, double precision, or string) based on the first letter of the variable name. Here's how it works.

Try this small program:

Variables beginning with these letters will be *typed* as integers →

```
100 REM ** DEFINING VARIABLE TYPES **
110 CLS
120 DEFINT A, X-Z
130 A = 1.25
140 Y = 3.14159
150 PRINT "A = ";A,"Y = ";Y
```

← A and X *through* Z

Line 120 tells the TRS-80 to treat all variables that begin with the letter A and those that begin with the letters X through Z as integer variables. The last *type definition* (X through Z) allows you to specify a range of variables to be of a particular type.

RUN the program and observe the results:

It worked! →

```
A = 1            Y = 3
READY
>-
```

The two variables contain only the integer portions of the numbers used in the program. The fractional portions were truncated because the DEFINT statement told the TRS-80 that A and Y were integer variables.

Similar statements exist for typing other variables:

Use	To Get These Variable Types
DEFINT	Inter
DEFSNG	Single Precision
DEFDBL	Double Precision
DEFSTR	Strings

The list of letters you want to use for integer, single precision, double precision, and strings are placed after these words

Enter and RUN this short program that uses all of the type definition statements:

```
100 REM ** TYPE DEFINITIONS **
110 CLS

200 REM **TYPE THE VARIABLE NAMES **
210 DEFDBL D
220 DEFSNG S
230 DEFINT I
240 DEFSTR Z

300 REM ** ASSIGN VALUES **
310 DPI = 3.141592653589
320 SPI = DPI
330 IPI = DPI
340 ZPI = STR$(DPI)

400 REM ** PRINT RESULTS **
410 PRINT "DPI = "; DPI
420 PRINT "SPI = "; SPI
430 PRINT "IPI = "' IPI
440 PRINT "ZPI ZPI = "; ZPI
```

π

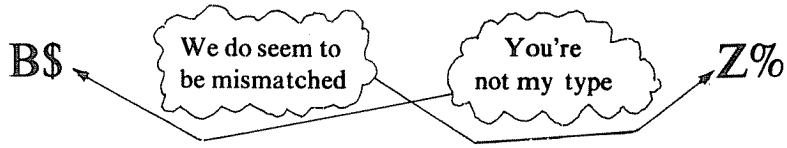
STR$ converts a number to a string

Your screen should show the following:

```
DPI = 3.141592653589
SPI = 3.14159
IPI = 3
ZPI = 3.141592653589
READY
>-
```

This is a string

Enough typing of variables. Just remember! When you type in programs and want a particular type of variable, use type definitions by typing the BASIC language words that tell the TRS-80 to type the variables the way you want them. Also remember, when you are typing not to make any errors in typing the type definitions. Got that? No?! Well, maybe typing variables is not the type of thing you like to do.



ERROR Handling on the TRS-80

The TRS-80 has a unique capability that gives you the chance to intercept errors that occur and create your own way of handling them. No one likes using a program where an inadvertent mistake causes the program to abort and stop, often with the display of some cryptic message that cannot be easily deciphered.

This short program demonstrates an abortive error condition:

Make N an integer variable →

```
100 REM ** ERROR DEMONSTRATION **
110 CLS
120 DEFINT N

200 REM ** ACCEPT AN INTEGER NUMBER **
210 INPUT "PLEASE ENTER A POSITIVE INTEGER";N
220 PRINT "OK, I GOT IT . . ."
230 PRINT "I WILL COUNT TO "; N
240 FOR I = 1 TO N: NEXT I
250 PRINT
260 GOTO 210
```

If you RUN the program, you will be asked to enter a positive integer. For integers from 0 to 32767, the program acknowledges the input and executes the FOR-NEXT loop N times. Try the program using several values such as 100, 200, and so on.

Now, try entering a value such as 100000. What happens? The program should stop and the screen shows:

OV? What's that mean? →

```
PLEASE ENTER A POSITIVE INTEGER? 1000000
?OV ERROR IN 210
READY
>-
```

Since N was *DEF*ined to be an integer variable, the largest value N can assume is 32767. The number 100000 is too big to go into N. The error message indicates this condition with the cryptic display:

OverFlow . . . of course!! → `?OV ERROR IN 210`

You need some way to detect that an error has occurred and have the program display a "compassionate" message without causing the program to terminate.

Add these few lines to the program and RUN the altered version:

```
12Ø ON ERROR GOTO 1ØØØ

1ØØØ REM ** ERROR HANDLING **
1Ø1Ø PRINT "I'M SORRY . . . THE NUMBER MUST
     BE LESS THAN 32767"
1Ø2Ø PRINT "PLEASE TRY AGAIN.": PRINT
1Ø3Ø RESUME
```

Two new Basic statements that you may not have used before

When you RUN this changed version, try entering 100000 and observe what happens:

```
PLEASE ENTER A POSITIVE INTEGER? 1ØØØØØ
I'M SORRY . . . THE NUMBER MUST BE LESS THAN 32767
PLEASE TRY AGAIN.

PLEASE ENTER A POSITIVE INTEGER? -
```

Your error message is displayed

The TRS-80 waits patiently for a number — the program does not stop

The statement at line 120 tells the TRS-80 that when an error occurs *anywhere* in the program control is to be transferred to line 1000. (Of course, you can choose whatever line number you want. The error handling does not *always* have to branch to line 1000. More on that later.) At lines 1010 to 1020 an error message is displayed, and at line 1030 a RESUME statement is encountered. The RESUME, in this form, tells the TRS-80 to continue the program *at the line where the error was encountered.* Clever!!

But, wait!! You can do even more than detect that an error has occurred. You can also determine what *line* of the program caused the error and actually detect what *type* of error has taken place.

Enter this next error-prone program and observe:

```
1ØØ REM ** MORE ERRORS **
11Ø CLS

2ØØ REM ** CLEAR SOME STRING SPACE **
21Ø PRINT "PLEASE ENTER THE NUMBER OF"
22Ø INPUT "BYTES OF MEMORY YOU WISH TO USE";
23Ø CLEAR N: PRINT

3ØØ REM ** ACCEPT DATA ITEMS **
31Ø PRINT "PLEASE ENTER THE ARRAY INDEX"
32Ø PRINT "NUMBER, AND THE STRING OF DATA"
33Ø INPUT "ITEMS (NUMBER,STRING):"; I,S$(I)
34Ø PRINT
35Ø GOTO 31Ø
```

This will clear N bytes of memory

Index

String goes here

RUN the program. The screen should clear and the request for the number of bytes should appear:

```
PLEASE ENTER THE NUMBER OF
BYTES OF MEMORY YOU WISH TO USE ?-
```

Let's say you enter 50 at this request; the program then displays:

```
PLEASE ENTER THE NUMBER OF
BYTES OF MEMORY YOU WISH TO USE ? 50

PLEASE ENTER THE ARRAY INDEX
NUMBER, AND THE STRING OF DATA

ITEMS (NUMBER,STRING):?-
```

At this point, you can enter pairs of numbers and strings. The number you enter is used to *index* the data into the S$ array. If you enter a one (1), the string of data that follows goes into S$(1). S$, however, is *not* DIMensioned, so values of I greater than 10 cause an error to occur. In fact, several places in this program can cause errors that stop the routine.

Here are three possible error conditions. Press the BREAK key, and reRUN the program. Enter the following responses and verify for yourself that these errors do occur.

Response	Screen Will Show

Enter 16000 for the number of bytes to CLEAR.	`PLEASE ENTER THE NUMBER OF`
	`BYTES OF MEMORY YOU WISH TO USE ? 16ØØØ`
	`?OM ERROR IN 23Ø`
	`READY` ← Out of memory

Enter 50 at the first input request: 11, "OOPS" at the second.	`PLEASE ENTER THE NUMBER OF`
	`BYTES OF MEMORY YOU WISH TO USE ? 5Ø`
	`PLEASE ENTER THE ARRAY INDEX`
	`NUMBER, AND THE STRING OF DATA`
	`ITEMS (NUMBER,STRING):? 11, "OOPS"`
	`?BS ERROR IN 33Ø`
	`READY` ← Bad subscript

Enter 10 at the first input request; 1,"12345678901" at the second.	`PLEASE ENTER THE NUMBER OF`
	`BYTES OF MEMORY YOU WISH TO USE ? 1Ø`
	`PLEASE ENTER THE ARRAY INDEX`
	`NUMBER, AND THE STRING OF DATA`
	`ITEMS (NUMBER,STRING):? 1,"123456789Ø1`
	`?OS ERROR IN 33Ø`
	`READY` ← Out of string space

Wow! What a lot of ways to get errors in such a small program. You could handle the errors by putting IF tests in the body of the program, but that method clutters up the program's flow. A simple, direct way to handle *all* these errors is to add an error handling routine.

Add these lines to the program and then try the same errors:

```
12Ø ON ERROR GOTO 9ØØ
24Ø ON ERROR GOTO 1ØØØ
9ØØ REM ** ERROR ROUTINE **
91Ø PRINT "THAT AMOUNT OF MEMORY IS"
92Ø PRINT "NOT AVAILABLE . . ."
93Ø PRINT "TRY ANY VALUE FROM Ø TO" MEM
94Ø PRINT: RESUME 21Ø
```

Note: Different line numbers are used

Tells how much memory is left

Return control at line 210

(ERR/2) + 1 = Error code**

```
1ØØØ REM ERROR IS AT LINE 33Ø **
1Ø1Ø REM ** CHECK FOR BS ERROR **
1Ø2Ø IF (ERR/2)+1 <>9 THEN 11ØØ
1Ø3Ø PRINT "INDEX NUMBER IS TOO BIG . . ."
1Ø4Ø PRINT "MUST BE FROM Ø TO 1Ø"
1Ø5Ø PRINT "PLEASE REENTER"
1Ø6Ø PRINT: RESUME 31Ø

11ØØ REM ** CHECK FOR OUT OF STRING ERROR **
111Ø IF (ERR/2)+1<>14 THEN 12ØØ
112Ø PRINT "THE LAST ENTRY EXCEEDED THE
     AMOUNT"
113Ø PRINT "OF AVAILABLE STRING SPACE . . ."
114Ø PRINT "PLEASE START OVER BY INCREASING"
115Ø PRINT "THE AMOUNT OF SPACE YOU NEED."
117Ø PRINT: RESUME 21Ø

12ØØ REM ** ANOTHER KIND OF ERROR OCCURRED **
121Ø PRINT "ERROR . . . PLEASE REENTER"
122Ø PRINT: RESUME
```

Error code 9 is BS error

Return to line 310

Error code 14 is out-of-string space error

Return to line 210

Return to line with error

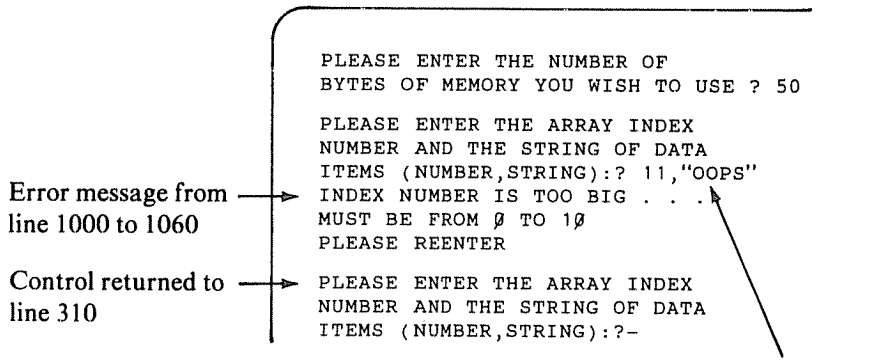
Here is what the screen shows as you enter the erroneous data items:

```
PLEASE ENTER THE NUMBER OF
BYTES OF MEMORY YOU WISH TO USE ? 16ØØØ
THAT AMOUNT OF MEMORY IS
NOT AVAILABLE . . .
TRY ANY VALUE FROM Ø TO 14615

PLEASE ENTER THE NUMBER OF
BYTES OF MEMORY YOU WISH TO USE ?-
```

Error message from lines 900 to 940

Control returned to line 210

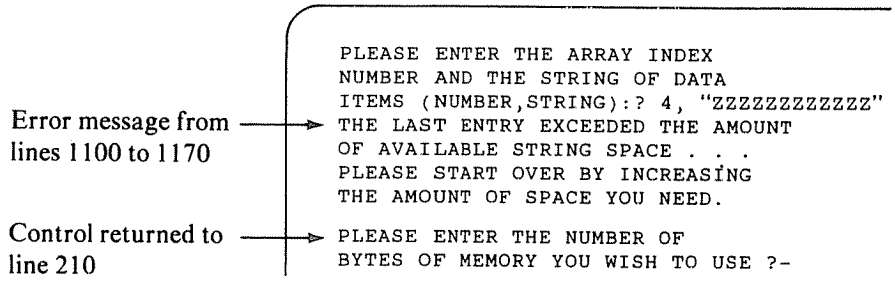
Enter 50 for this input and then try the second error:



S$(11) would normally cause a ?BS error in line 310. The ON ERROR GOTO routed the program to the error routine

**For a list of error codes, see Appendix C.

Now, enter valid index numbers but with long strings of data until the last error condition occurs:



The error routines handle only out-of-memory (OM), bad subscript (BS), and out-of-string space (OS) errors. It's possible to have other kinds of errors occur, especially in line 330. Lines 1200 through 1220 provide a general error message for any other cases and RESUMEs the program at the input line where the error occurred. See if you can discover an error that will cause the message at line 1210 to be displayed.

Other ERRORS I Have Known

You have just been introduced to the error handling capabilities of the TRS-80. You have used the ON ERROR GOTO, RESUME, and ERR features. RESUME was used in two ways: 1) To return the program to the line where the error occurred, and 2) to return the program to a different line by specifying a line number. RESUME has one more possible use — to ignore an error or continue program execution. By putting the words RESUME NEXT in your program's error-handling routines, control is given back to the *next* line in the program after the point where the error happened.

So, RESUME can be used in three ways:

Usage	Action
RESUME or RESUME 0	Resumes program at the statement containing the error
RESUME 100	Resumes at line 100
RESUME NEXT	Resumes at line *after* the point where the error occurred

The TRS-80 also has one other variable name — ERL — that contains the *line number* of the line that caused the error condition. ERL can be used in IF-THEN statements such as:

```
9Ø5 IF ERL<>23Ø THEN 1ØØØ
```

ERROR Summary

You can use a combination of ON ERROR GOTO, RESUME, ERR, and ERL statements and variables to build your own error routines. The ON ERROR GOTO statements tell the TRS-80 the location of the error routines. Each error routine can contain one or more RESUME statements that return control to the main program at the location you want control returned. You can use ERL and ERR variables to tell you where the errors occurred and what particular error conditions were encountered.

The rest of the book shows examples (where appropriate) of the use of error processing within the program presented. The TRS-80 error handling features give you great flexibility in the way *you* want error messages and error recovery to work within your programs.

Summary

Two major capabilities of your TRS-80 are:

- The use of *type definitions* and symbols.
- The use of error handling routines.

The TRS-80's BASIC language has four classes of variables: integers, single precision, double precision, and strings. You can tell the TRS-80 what variable type you want to use in two ways: 1) put one of the following symbols (%, !, #, $) after the variable name, or 2) use a type definition (DEFINT, DEFSNG, DEFDBL, DEFSTR).

In learning to use these methods of expressing variable types, you saw that each type uses different amounts of memory and that significant memory savings can be achieved by using the appropriate variable type. You also discovered that the TRS-80 has several unique features built into the BASIC language to facilitate error processing. You can build your own error routines and can intercept the normal error messages that the TRS-80 generates. You have the program display messages of *your* choice; messages that state what you want to say to the user. The error handling statements and variables (ON ERROR GOTO, RESUME, ERL, ERR) can prevent the program from stopping unexpectedly because of an error condition.

Now try your hand at the self-test that follows as a brief review of what was discussed in this chapter.

Self-Test

1. Match the correct symbol from the second column with the names in the first column, as they are used in the TRS-80 to specify the *type* of variable within a program:

(a)	Integer_____	1. #
(b)	Single Precision_____	2. $
(c)	Double Precision_____	3. %
(d)	String_____	4. !

2. Which of the following four *type definition* declarations

   DEFINT DEFSNG DEFDBL DEFSTR

   are used to specify that variables beginning with certain letters are to be:

   (a) integer variables? _____
   (b) string variables? _____
   (c) double precision? _____
   (d) single precision? _____

3. If you use the following statement in a program,

```
1ØØ DEFINT A,D,M-Q
```

which of these variables will be integer valued?_____

(a) COSTS (b) DEBT (c) TAXES (d) ASSET(5,3) (e) PARTS

4. How many bytes will you save if you use DIM A%(999) in place of DIM A!(999) in a program?

(a) 1000 bytes (b) 2000 bytes (c) 4000 bytes (d) none

5. What are the number of bytes required for each of the following simple variables?

(a) A%_____ (b) A!_____ (c) A#_____ (d) A$_____

6. A mathematical constant called e has a value of:

e = 2.7182818284590452 . . .

If the following assignments are made, what will the contents of each variable contain if they were PRINTed?

Assignment	Contents
E# = 2.182818284590452	a._____
E! = E#	b._____
E% = E#	c._____
E$ = STR$(E#)	d._____

7. The following assignment was made and the result PRINTed. Explain the results.

```
>A# = 1/3
READY
>PRINT A#
  .3333333432674408
READY
>-
```

8. If you want to know how much memory is free for string storage, what BASIC function would you use?

9. What are the four key statements and variables used in developing your own error handling routines?

a._____ b._____ c._____ d._____

10. Describe what appears on the screen when these two PRINT statements are executed:

    a. PRINT ERL _____

    b. PRINT (ERR/2)+1 _____

Answers to Self-Test

1. a. 3(%) b. 4(!) c. 1(#) d. 2($)
2. a. DEFINT b. DEFSTR c. DEFDBL d. DEFSNG

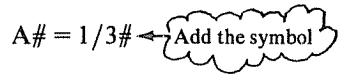
3. Variables b, d, and e are integer variables.
   Variables a and c would not be integer valued unless there was another type statement in the program (DEFINT C,T) that told the TRS-80 to make them integers.

4 b. 2000 bytes (1000 elements times a savings of 2 bytes/element)

5. a. 5 bytes b. 7 bytes c. 11 bytes d. 6 bytes plus 1 byte for each character in the string.

6. a. 2.182818284590452 b. 2.182818 c. 2
   d. 2.182818284590452

7. The last part of the double precision result was not assigned. The constant 1/3 was calculated as a single precision value, and then assigned to A#. To correct this problem, the assignment should read:



8. You would use FRE(A$) or FRE("DUMMY"). The argument can be any string value (it is a "dummy" argument). FRE can be used in calculations and PRINT statements. It returns the value of the amount of string space still available for use by the program.

9. a. ON ERROR GOTO b. RESUME c. ERR d. ERL

10. a. The line number of the last program line where an error occurred.
    b. The error code of the last error to occur.
    These two variables are valuable in constructing your own error handling routines.

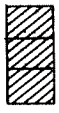
CHAPTER TEN

Graphics and Animation

Earlier, in chapter 3, you looked at some of the TRS-80's graphic abilities using the SET command, POKE command, CHR$ function, and string operations. You found that some BASIC statements put graphic characters or strings of information on the screen quickly. Some, like SET, do so more slowly but let you control a finer pattern of graphics on the display. To review, let's build a small program that compares several ways of putting the same information on the TRS-80's display.

The Race Is On

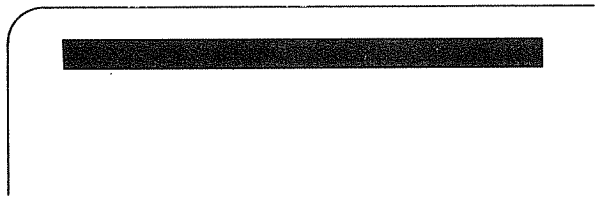
To start, enter and run this small program:



SET, three small rectangles to form:

```
100 REM ** GRAPHICS COMPARISON **
110 CLS

200 REM ** USING THE SET COMMAND **
210 FOR X = 0 TO 127
220   SET(X,0): SET(X,1): SET(X,2)
230 NEXT X

600 REM ** WAIT AND THEN REDISPLAY **
610 PRINT@896, "PRESS ANY KEY TO CONTINUE"
620 IF INKEY$="" THEN 520 ELSE 110
```

Does your screen have a "bar" of light across the top?



```
PRESS ANY KEY TO CONTINUE
```

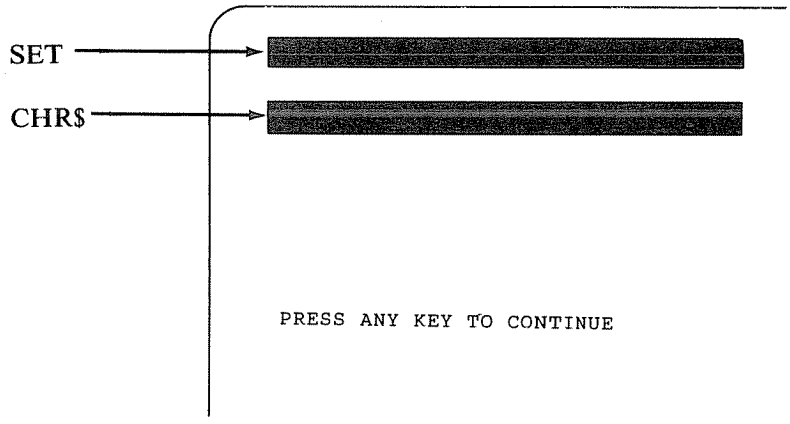
The SET commands produce a bar across the top of the screen. Three rows of 128 tiny rectangles are "turned on" by the SET statements. Press a key and watch the operation closely. As it is being drawn, the front edge of the bar seems a bit wavy. There are moments when you can see each SET statement at work. The small rectangle on the first row is SET, then the one on the second row, and, finally, the one on the third row.

Now, add the next segment of the program and compare the results. Press the Break key and enter:

Also produces the character 

```
3ØØ REM ** USING CHR$ **
31Ø FOR X = Ø TO 63
32Ø   PRINT@256+X, CHR$(191);
33Ø NEXT X
```

When this expanded program is RUN, the race *is* on. At the end of the first lap, the screen shows:



The SET commands and PRINTing CHR$(191) do the same thing — put a bar across the screen. The difference is that the PRINT statement does the operation *faster*. Press a key and observe the two bars being drawn again. No doubt about it, the PRINTing of CHR$ is fast.

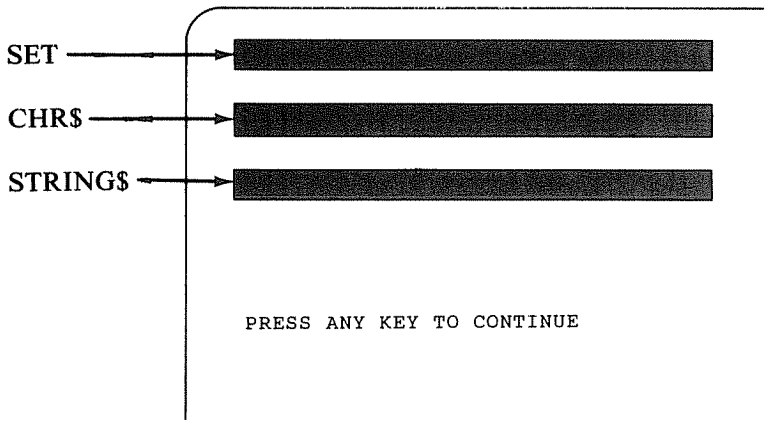
Is there yet a faster way? Yes, there is. Press the Break key to stop the program and add these lines for the last entry into the race.

Remember, you have to CLEAR extra string space for more than 50 characters

```
4ØØ REM ** USING STRING$ **
41Ø CLEAR 64
42Ø PRINT@512, STRING$(64,191);
```

Produces 64 across screen

After one race is run, the screen shows:



Using STRING$ places the bar on the screen much more quickly than any of the other methods. Press a key and observe how fast the three bars are created. The STRING$ version goes on in the blink of an eye.

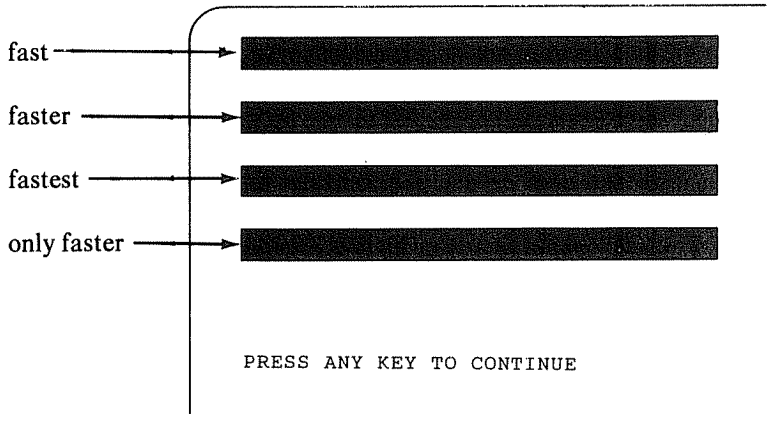
The race is now over and you have a clear winner . . . uh oh! Here comes one more entry around the bend. You thought the race was complete, but there is still one more way to put the bar across the screen. The entry is getting closer. It looks like slowPOKE. Yes, it is slowPOKE; only slowPOKE is *not* so slow.

Observe by runnning this program:

How fast is one?

```
5ØØ REM ** USING POKE **
51Ø FOR X = Ø TO 63
52Ø   POKE 16128+X, 191
53Ø NEXT X
```

When you enter and RUN the program, your screen should show:



Was POKEing the bar across the screen faster than using STRING$? Press a key and look once more. Yep, STRING$ is still the fastest way to put the bar on the screen. POKE is also fast, but doesn't seem to be any quicker than CHR$.

Here is a complete listing of the graphic comparison program:

```
100 REM ** GRAPHICS COMPARISON **
110 CLS

200 REM ** USING THE SET COMMAND **
210 FOR X = 0 TO 127
220   SET(X,0): SET(X,1): SET(X,2)
230 NEXT X

300 REM ** USING CHR$ **
310 FOR X = 0 TO 63
320   PRINT@256, CHR$(191);
330 NEXT X

400 REM ** USING STRING$ **
410 CLEAR 64
420 PRINT@512, STRING$(64,191);

500 REM ** USING POKE **
510 FOR X = 0 TO 63
520   POKE 16128+X, 191
530 NEXT X

600 REM ** WAIT AND THEN REDISPLAY **
610 PRINT@896, "PRESS ANY KEY TO CONTINUE"
620 IF INKEY$="" THEN 620 ELSE 110
```

fast → 210
faster → 320
fastest → 410
POKE along → 520
Wait!! → 610

Let's go with the winner for a while and take the STRING$ function to look at other graphic operations that can be done quickly.

Splitting the Screen

Using STRING$, the placement of graphic characters is fast enough to let you do several graphic operations that appear to be happening at the same time.

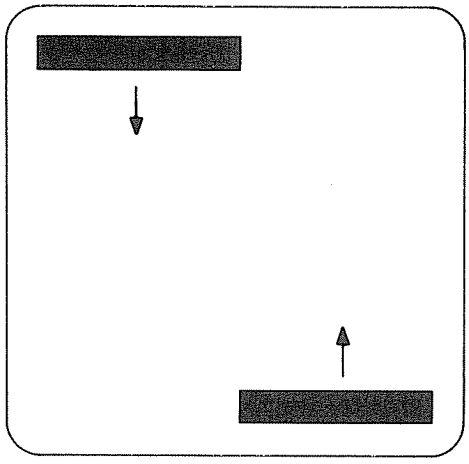
Enter the following lines into your TRS-80 and RUN them:

```
NEW

100 REM ** SPLITTING THE SCREEN **
110 CLS

200 REM ** TOP TO BOTTOM/ BOTTOM TO TOP **
210 FOR Y = 0 TO 14
220   PRINT @0 + Y*64, STRING$(32,191);
230   PRINT @928 - Y*64, STRING$(32,191);
240 NEXT Y

300 REM ** WAIT AND REDISPLAY **
310 IF INKEY$ = "" THEN 310 ELSE 110
```

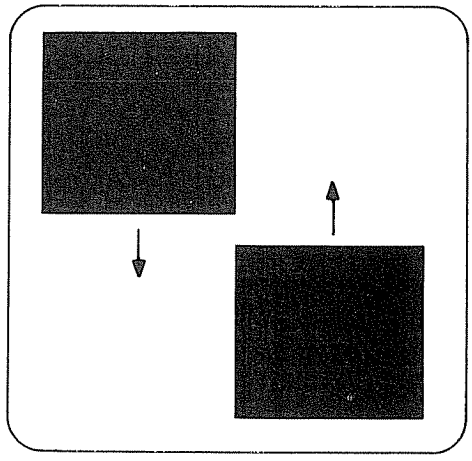
Left side of screen → 220
Right side → 230
Wait → 310

When this program is RUN, the screen should show the following sequence of events:

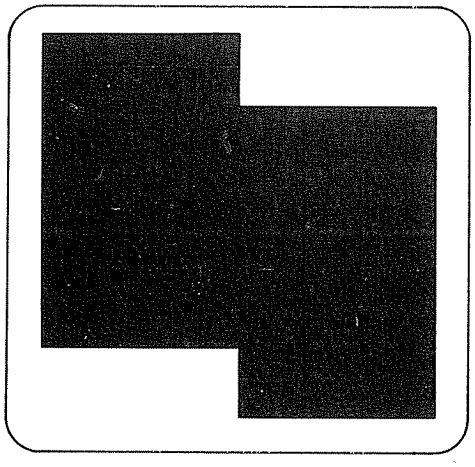
At the start . . .



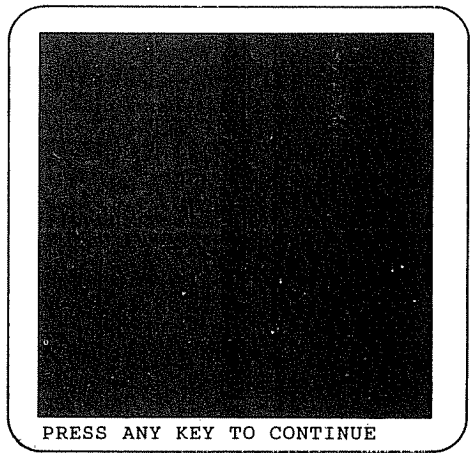
Later . . .



Even later . . .



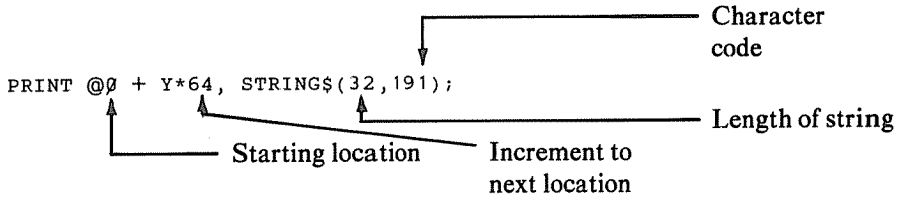
At the end.



Press the space bar several times in succession. The screen empties and refills within seconds. The left half fills from the top down; the right from the bottom up. The actions appear to occur at the same time. The end result is like opening a "window" into your TRS-80. The white area surrounded by the black border gives you the illusion of looking into a "window." Hmm . . . is it possible to make the window any size and put it anywhere?

What Light through Yonder Window . . .

The use of PRINT@ and STRING$ lets you place a string of characters anywhere on the screen and control the size of the string within STRING$. In the last program, the statement:



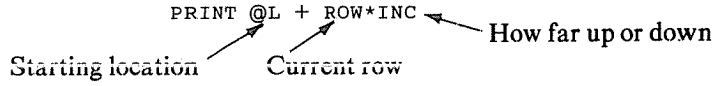
lets you specify completely where the string is to be placed (@0 + Y*64), how long it is to be (32), and what the displayed character is to be (191). With a little modification, this one PRINT statement can become a generalized shape generator or window maker. Before building the program, let's examine the PRINT statement that is the heart of the routine and see how it is constructed.

Suppose, to start, that you want to begin your window anywhere on the screen. This requirement means a variable must follow the @ symbol:

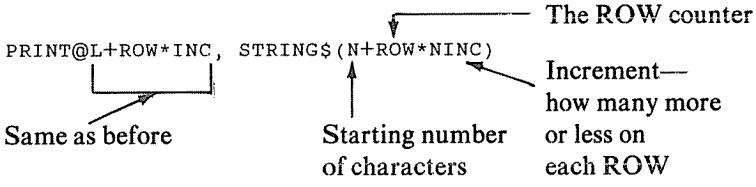
```
PRINT @L
```

L is for the starting *L*ocation of the window

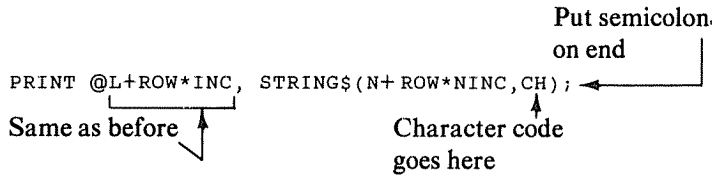
To put parts of the window down the screen, the starting location must be changed. Add a variable that changes based on a FOR-NEXT loop (ROW) and an increment that you want to move up or down the screen (INC) to totally modify the starting location.



The PRINT statement now positions the strings of data on the screen. You can add the STRING$ function, but let's also make it as flexible as possible. Let the number of characters to be printed have a beginning length and let that length vary with each ROW that is displayed. If you use N for the *n*umber of characters and NINC for the increment that changes that number from ROW to ROW, then the PRINT statement looks like this:



To make the statement completely general, let the displayed character be a variable (CH):



OK? It may seem like a lot of effort, but remember, this *one* statement is going to do all the work. Now let's build the rest of the program around the PRINT statement and see what can be done.

Enter the following:

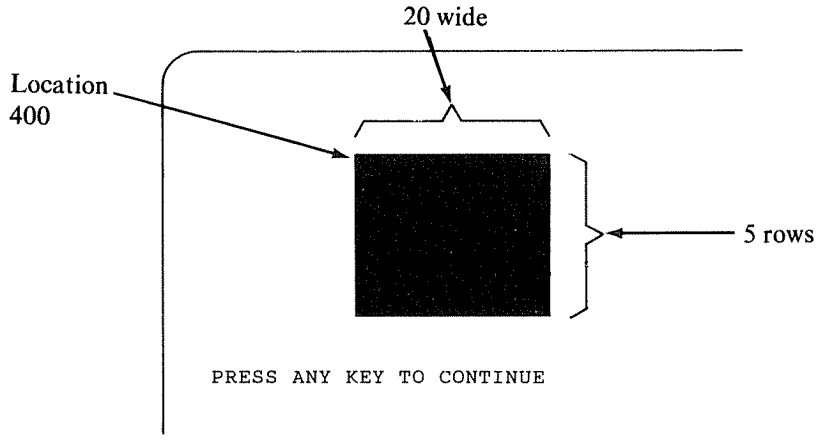
5 rows; starting at location 400; moving down one row each time (64); with 20 characters each row (NINC=0); character code 191

```
NEW

100 REM ** SHAPE MAKER **
110 CLS

200 REM ** ASSIGN INITIAL VALUES **
210 NROWS=5: L=400: INC=64
220 N=20: NINC=0: CH=191

300 REM ** DISPLAY SECTION **
310 FOR ROW = 0 TO NROWS-1
320   PRINT@L+ROW*INC,STRING$(N+ROW*NINC,CH);
330 NEXT ROW

500 REM ** WAIT AND REDISPLAY **
510 PRINT @896, "PRESS ANY KEY TO CONTINUE"
520 IF INKEY$= "" THEN 520 ELSE 110
```

The work horse

That's all there is to the program. RUN it.

Does this display appear on your screen?



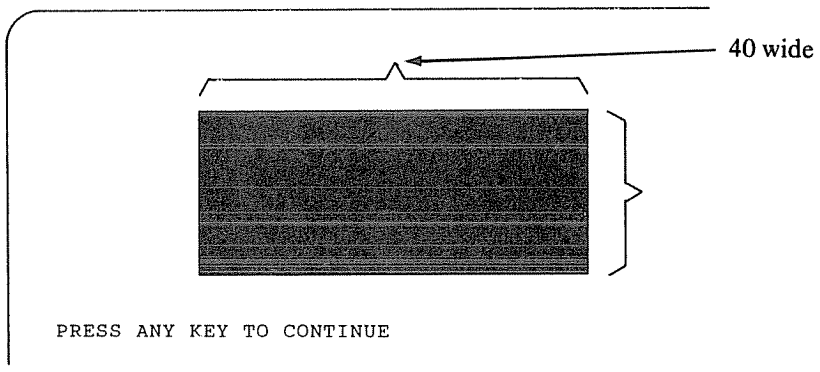
Simple! Want to make the window wider? That is easy too.

Enter this line and RUN the program:

```
220 N=40: NINC=0: CH=191
```

The "window" will be twice as wide

Does your screen now show:



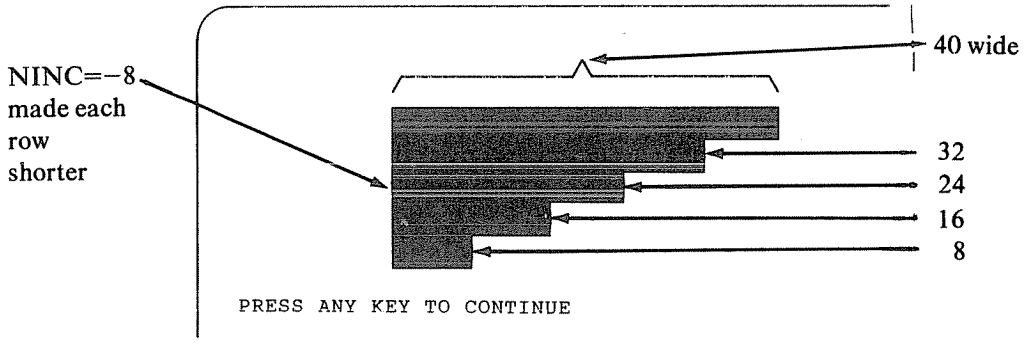
Changing the value of N from 20 to 40 causes a rectangular "window" to appear on the screen. In a similar fashion, you can alter any of the variable values in lines 210 and 220, and other shapes appear. Pressing any key except the Break key causes the shape to be redisplayed.

Try this change:

```
220 N=40: NINC=-8: CH=191
```

Hmmm . . . a *minus* eight! Wonder what that will do . . . ?

If you made the last change and have run the altered program, your screen should show:



Interesting! You do have a shape maker, but you also need a simple way to change the variable values so you can have the new shapes drawn quickly. Let's alter the way the variables are assigned values and use INPUT statements to change values.

A New Shape Maker

Here is a much more flexible version of the shape maker. The program is altered in several ways to add the flexibility INPUTting the values directly. The alterations are worth the extra effort, as you will see. Here is the new program:

```
NEW
100 REM ** NEW SHAPE MAKER **
110 CLS
120 CLEAR 1000
200 REM ** ACCEPT THE INPUT VALUES **
210 PRINT @0, "CHANGE VALUES(KEYS 1 TO 6):CONTINUE(OTHER KEYS)
220 PRINT @64,"1(NROWS),2(LOC),3(INC),4(NCHR),5(NINC),6(CHR)
230 A$=INKEY$: IF A$="" THEN 230 ELSE I=VAL(A$)
240 IF I=0 OR I>6 THEN CLS: GOTO 410
250 REM ** CLEAR THE FIRST TWO ROWS **
260 PRINT @0, STRING$(128," ");
270 PRINT @0,"THE CURRENT VALUE FOR VARIABLE" I "IS" A(I-1)
280 PRINT @64, "ENTER THE NEW VALUE";
290 INPUT A(I-1)
300 GOTO 210
400 REM ** DISPLAY SECTION **
410 FOR ROW = 0 TO A(0)-1
420   PRINT @A(1)+ROW*A(2), STRING$(A(3)+ROW*A(4),A(5));
430 NEXT ROW
440 GOTO 210
```

Make room for strings

This part handles the INPUTS

The array A is being used to store values A(0) through A(5)

Enter and RUN this program. The first thing you see should be:

```
CHANGE VALUES (KEYS 1 TO 6);CONTINUE(OTHER KEYS)
1(NROWS),2(LOC),3(INC),4(NCHR),5(NINC),6(CHR)
```

Start location

Number of rows

Number of characters

How far up or down

Character code

How many more or less each row

Press the number one (1). What happens? The screen clears and then shows:

```
THE CURRENT VALUE FOR VARIABLE 1 IS Ø
ENTER THE NEW VALUE?-
```

Enter the value 5 and press the Enter key. The screen clears, and the first display returns:

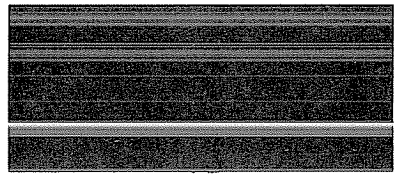
```
CHANGE VALUES(KEYS 1 TO 6);CONTINUE(OTHER KEYS)
1(NROWS),2(LOC),3(INC),4(NCHR),5(NINC),6(CHR)
```

Enter the following information:

PRESS	Then Type	
2	400	Press ENTER after the values are typed
3	64	
4	20	
6	191	
ENTER		

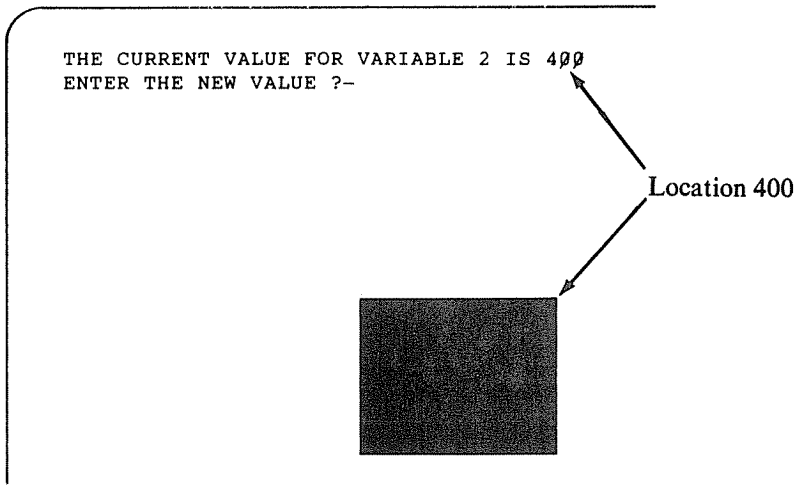
When you press the Enter key the last time, the screen shows:

```
CHANGE VALUES(KEYS 1 TO 6);CONTINUE(OTHER KEYS)
1(NROWS),2(LOC),3(INC),4(NCHR),5(NINC),6(CHR)
```

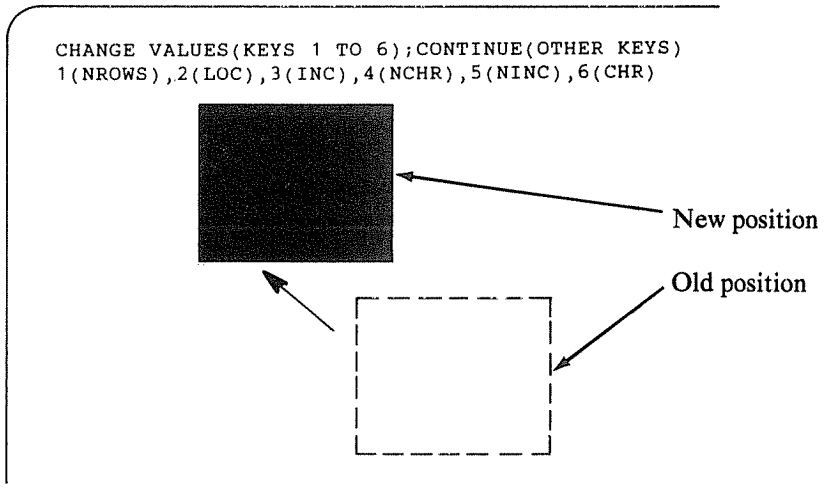


Aha! Your "window" is back. Now move the "window". Press key 2.

The screen displays:

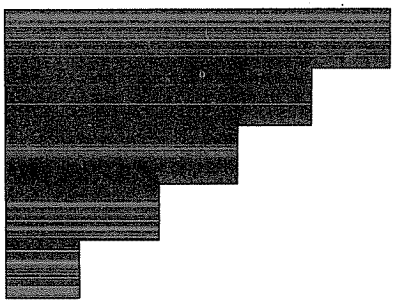


Enter 200, press the Enter key, and then press the Enter key again. The window moves up and to the left, like so:

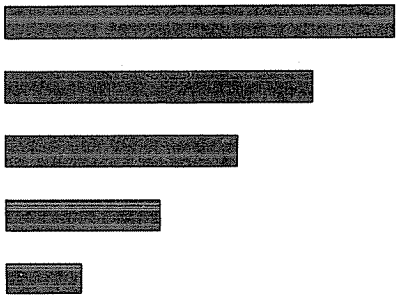


Experiment with other value changes. Change the location back to 400 (variable 2) and change variable 5 to a minus 4.

The sawtooth shape appears:



Change variable 5 back to zero, and change variable 6 to character code 188. Is this what you see?



Now, once again set variable 5 to a minus 4.
Does the "bar chart" appear?

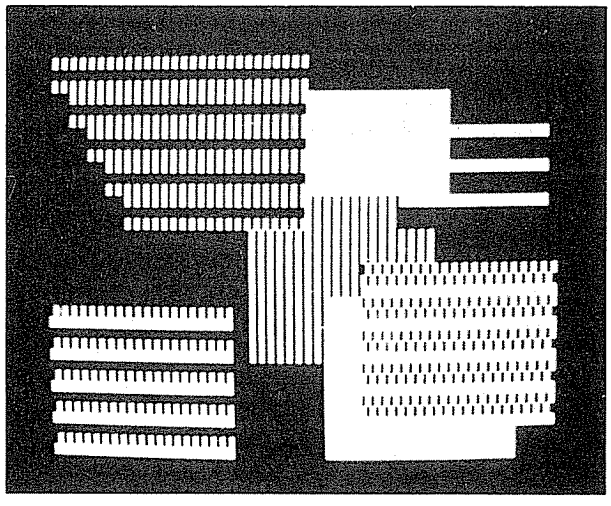
Experiment with other changes. Alter the character code while keeping the same shape. What do you discover? Is it possible to make these two shapes? (Hint: You must set variables 3 *and* 5 to specific values to get these two shapes.)

The TRS-80 Electronic Easel

The TRS-80's graphic abilities are fun to use. Make this one small change to the shape maker program you have been using, and then sit back for your first electronic art lesson:

```
24Ø IF I=Ø OR I>6 THEN 41Ø
```

Now the shape maker will *not* erase the shapes you put on the screen. You can put one shape on the screen, move it to another location, and both are displayed. You can then add a third shape, alter the character codes, and put as many different shapes as you want on one screen. Experiment with the program. Here is a picture of a shape we created:



If you want to remove a shape from the screen, use character code 32 (space). You can also use this code to create "holes" in your drawings. Try other codes also, such as the numbers 48 through 57, and the letters 65 through 90. Have fun with the electronic easel.

Shake, Rattle, and Rumble

The entire discussion on the shape maker programs was designed to show you how a single routine can be used in variety of ways. The shape maker is like a tool in your toolbox of routines that you can pull out and incorporate into other programs you are building.

Here is another such routine that can be used to simulate being hit by cannon fire or being caught in an earthquake. (Art Canfield of Cybernautics Software uses something like this in his game *Taipan.*)

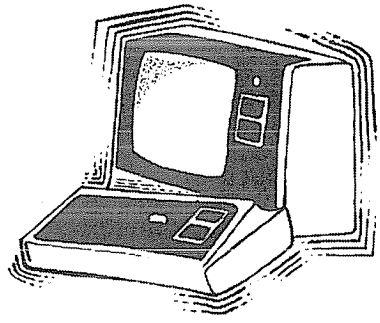
Enter this program and RUN it to see what happens:

```
NEW

100 REM ** THE SHAKER PROGRAM **
110 CLS

200 REM ** PUT A SHAPE ON SCREEN **
210 FOR ROW = 0 TO 4
220   PRINT @400+ROW*64, STRING$(20,188);
230 NEXT ROW

300 REM ** MAKE IT SHAKE **
310 FOR I = 1 TO 20
320   PRINT CHR$(23)
330   REM ** WAIT A BIT **
340   FOR J=1 TO 10: NEXT J
350   PRINT CHR$(28)
360 NEXT I

400 REM ** WAIT AND REDISPLAY **
410 IF INKEY$="" THEN 410 ELSE 110
```

5 bars at location 400

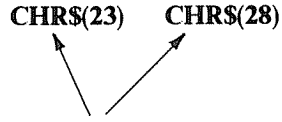
What's this?

The image is placed on the screen and the screen then appears to go crazy for a moment. Press a key (except the Break key), and the shaking repeats. Great! Now you have a routine to shake things up a bit.



Big and Small

The shaker program that you just examined makes two new uses of CHR$:



To demonstrate what these two character codes are actually intended to do, enter and run the following program:

Easy on the typing fingers →

```
NEW

100 REM **SMALL TO BIG**
110 CLS
120 PRINT@448, "YOUR TRS-80 WILL SERVE
    YOU WELL"
```

The message in the PRINT statement appears on the screen:

The message →

```
YOUR TRS-80 WILL SERVE YOU WELL
READY
>_
```

Now, add the following line to the program:

```
115 PRINT CHR$(23)
```

What happens when you RUN this altered program? Yes, the message appears, but in *big* letters. CHR$(23) tells the TRS-80 to display 32 characters across the screen where there are normally 64 characters. The characters appear twice as big with CHR$(23).

Great! You can make your program messages easy to read by using CHR$(23). What about CHR$(28)? If you PRINT CHR$(28), the TRS-80 goes back to displaying smaller characters.

Type the statement in the Immediate Mode and observe the screen:

```
PRINT CHR$(28)
```

Return to regular-sized characters

Your screen should show:

```
READY
>-

YOUR TRS-80 WILL SERVE YOU WELL
READY
> PRINT CHR$(28)
```

Message is there, in small letters

Where did all the spaces come from?

CHR$(28) tells the TRS-80 to go back to 64 characters per line, and it also homes the cursor (puts it in the upper left corner of the screen). The READY message appears at the home position in the regular small letter format.

Hold it! The messages that were on the screen before the PRINT CHR$(28) was executed have spaces *between* each letter. Why so? Well, when CHR$(23) is used the TRS-80 takes up two positions from the 64-character line to make one position in the 32-character line. When the process is reversed, each of the second-character positions in the 64-character line are empty. This feature offers some interesting possibilities to vary your displays. *But, one word of caution!!* When using CHR$(23), every PRINTed message must be placed on the screen at an *even numbered* location.

Change line 120 to what follows and look at the results:

```
120 PRINT @449, "YOUR TRS-80 WILL SERVE YOU WELL"
```

Odd numbered location

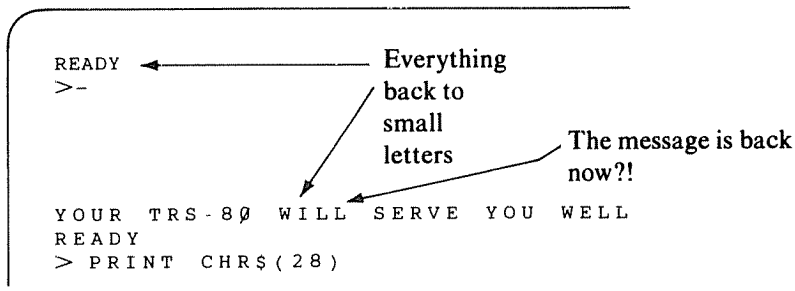
RUN this program. What do you see? Nothing? Yes, you see nothing.

This will be in big letters

```
READY
>-
```

No message!!

Type PRINT CHR$(28) in the Immediate Mode. Does your screen show this?



The message is still there, but the characters are in each of the second-character positions. When CHR$(23) is used, the TRS-80 takes the characters in each of the first positions (they are spaces) and expands them into the larger character format. In this case, you get a row of bigger *spaces*. Oh, well!! Change the PRINT location back to 448 or some other even-numbered value and this problem disappears (or reappears, as the case may be).

Billboard Time

If you have changed line 120 so that the PRINT location is 448, your program looks like this:

Makes big characters →

```
100 REM **SMALL TO BIG**
110 CLS
115 PRINT CHR$(23)
120 PRINT @448, "YOUR TRS-80 WILL SERVE
    YOU WELL"
```

Let's add a few lines to turn this program into a flashing billboard display:

Wait! →

```
130 FOR I = 1 TO 300: NEXT I
140 PRINT CHR$(28)        <- Big letters
150 FOR I = 1 TO 300: NEXT I
160 GOTO 115              <- Back to large letters
```

Wait again →

The CHR$(23) at line 115 puts the message in large letters. CHR$(28) at line 140 sets everything back to 64 characters per line. The message appears in smaller letters but with spaces between each character. The delays at lines 130 and 150 hold each version of the message on the screen for a while. The result: a *flashing* billboard message.

RUN the program, and see it for yourself. As you watch the message flash between the two character sizes, think back to the Shaker Program. Can you now see why that earlier program seemed to make objects on the screen "shake?" The program was making several rapid shifts from 32 characters to 64 characters and back again. The effect produced was like watching the object being vibrated or hit by cannon fire. The use of CHR$ to "shake" the screen was probably not meant to be an intended use of the function. Nevertheless, it *is* an effective way to simulate being "shaken," and you may find many more uses for this feature in your programs.

New York, New York . . .

Watching the flashing billboard may remind you of other forms of billboard signs, like the ones in New York's Times Square. Some signs flash; others display a message a few letters at a time and move the message past your vision from right to left. Can the TRS-80 be made to display one of these "endless" messages? Of course it can.

Enter this program to see the TRS-80 turn into a billboard display sign:

```
100 REM ** TRS-80 BILLBOARD DISPLAY **
110 CLS
120 A$ = "      YOUR TRS-80 WILL SERVE YOU WELL . . .  "
```

One space

```
200 REM ** ROLLING DISPLAY **
210 I = 1
220 PRINT @480 MID$(A$,I,6)
230 FOR J = 1 TO 50: NEXT J
240 I = I + 1
250 IF 1>LEN(A$) THEN 210 ELSE 220
```

Start with one short wait!

Displays 6 characters at a time

Check for end of message

RUN the program and observe with delight what appears. Here is a representation of what you should see:

First –	Y
Next –	YO
Next –	YOU
Next –	YOUR
Next –	YOUR
Next –	YOUR T
Next –	OUR TR
etc.	etc.

Message is moving in this direction ←

This animation effect is difficult to show on paper. Run the program.

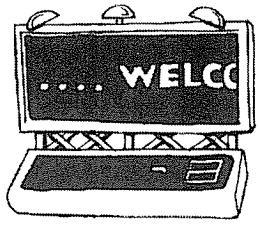
The partial message moves from right to left past your vision and disappears off the left side. The entire message is displayed and then begins again. This technique is a clever way to pack a large message into a small "window" on the screen. (The Carlston brothers of Broderbund Software use examples of this horizontal scrolling in their programs the *Galactic Trilogy*.) The billboard sign is being displayed in small letters. Would you like a bigger sign?

OK, change line 110 to read:

```
11Ø CLS: PRINT CHR$(23)
```

Big letters

RUN the altered program and your sign should appear in large letters. You can use this routine in many ways in your programs.



Billy the Goat

Up to now, you have seen several techniques for displaying graphics. You have not used the SET and RESET commands, but rather have relied on the faster CHR$ and STRING$ functions. However, there are places where SET and RESET can be used effectively. Since they access the small rectangles on the screen, you can use them in situations where CHR$ and STRING$ may not be effective. The next example ultimately combines STRING$ with SET and RESET.

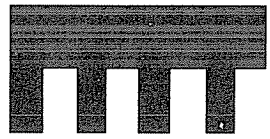
Enter and RUN this program:

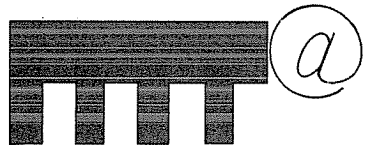
```
NEW

1ØØ REM ** BILLY GOAT **
11Ø CLS
12Ø CLEAR 2ØØ

2ØØ REM ** CREATE THE GOAT **
21Ø A$ = STRING$(4,156)+"@"

22Ø REM ** WALK THE GOAT **
23Ø FOR X = Ø TO 59
24Ø   PRINT @Ø, STRING$(X,32)+A$
25Ø NEXT X
```

This is a goat?

You used this technique in chapter 3

When you ran the program did "goat" appear and walk across the screen? Oh, it appeared, but it *ran* across the screen. Let's slow it down then.

The character 156 is  . Four of these look like: 

When you add the @ sign, you get a "goat." 

Type these lines into the program:

Slow the goat →
```
25Ø FOR I = 1 TO 5Ø: NEXT I
26Ø NEXT X
```

There! Does the goat "walk" now? It does, but its gait is rather stiff-legged. So, let's make another modification to the program to move the goat's "legs."

Enter these program lines into the TRS-80:

Randomly set and reset the goat's "legs" →
```
25Ø   FOR I = 1 TO 2
26Ø     R = RND(8)-1
27Ø     RESET(X*2+R,2)
28Ø     R = RND(8)-1
29Ø     SET(X*2+R,2)
3ØØ   NEXT I
31Ø NEXT X
```
← Leg position is location of goat's tail(X), times two (X×2), plus the random number R(0 to 7).

What happens when you RUN this program? The goat now "walks" across the screen, and its legs are "moving" on their own, not just with the movement of the body.

Thus, SET and RESET can be combined with other graphic statements to produce special effects on the screen. As you remember, CHR$ and STRING$ display six small rectangles at once. SET and RESET work with only one of the screen's tiny rectangles. A normal line has 64 character positions, but 128 SET/RESET positions. That difference is why lines 270 and 290 in the program multiply the position of the "goat's tail" (X) by two (X×2). A humorous action occurs if you change each line to X+R. Try it and see what occurs.

Summary

This chapter has expanded your ability to use your TRS-80's graphic powers. You discovered that the STRING$ function is the fastest way to put strings of characters on the screen. You found that you could split the display area into several pieces and made it appear as if you were doing something in each area at the same time.

You also developed a shape generator and shape maker program and then modified it into an electronic easel.

Finally, you explored several small routines to simulate "shaking" the screen. The CHR$ functions changed the size of the characters that the TRS-80 generated. You turned the TRS-80 into a flashing billboard and one that displayed an endless message. And, of course, you played with Billy the Goat.

Time now for a few exercises (both physical and mental). After you have flexed your muscles, move on to the self-test at the end of this chapter and flex your mind. See you again in chapter 11.

Self-Test

1. Which of the following statements and functions put characters on the screen the fastest?

   _____

   (a) SET (b) CHR$ (c) STRING$ (d) POKE

2. Write a statement to put a solid bar on the screen from locations 448 to 575 (two rows). First, type: `CLEAR 128`

   `PRINT @_____`

3. Write a set of statements to quickly place a *vertical* bar on the screen, beginning at location 30 and going to the bottom of the screen. Make the bar 10 characters wide.

   `CLS: FOR Y = Ø TO 15: PRINT@_____: NEXT Y`

4. Write a set of statements to quickly place a "stair step" set of bars 10 characters wide from the upper left corner to the bottom of the screen. Make each bar shift one location to the right from the preceding bar.

   `CLS: FOR Y = Ø TO 15: PRINT @_____: NEXT Y`

5. Describe what CHR$(23) and CHR$(28) do when PRINTed.

   _____

   _____

   _____

6. Write a small program to "flash" from big to little letters the message:
   ATTENTION !! ATTENTION !!

7. Write a program to horizontally scroll an "arrow" from *left to right* in a small "window" near the center of the screen.

8. Describe what happened in the Billy the Goat Program when the SET and RESET parameters were changed to X+R.

Answer to Self-Test

1. (c) STRING$ is the fastest way to place characters on the screen.

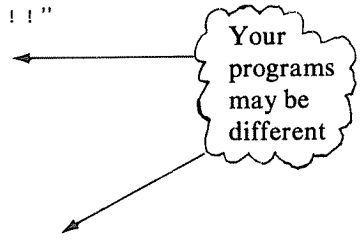
2.
```
PRINT @448, STRING$(128,191);
```

3.
```
CLS: FOR Y = Ø TO 15: PRINT @ 3Ø+Y*64, STRING$(1Ø,191);:NEXT Y
```

4.
```
CLS: FOR Y = Ø TO 15: PRINT @Ø+Y*65, STRING$(1Ø,191);:NEXT Y
```

5a. CHR$(23) tells the TRS-80 to use 32 characters per line.
5b. CHR$(28) tells the TRS-80 to use 64 characters per line. CHR$(28) also homes the cursor (moves it to the upper left corner of the screen).

6.
```
1ØØ REM ** FLASHING MESSAGE **
11Ø CLS
12Ø PRINT CHR$(23)
13Ø PRINT @448, "ATTENTION !! ATTENTION !!"
14Ø FOR I = 1 TO 3ØØ: NEXT I
15Ø PRINT CHR$(28)
16Ø FOR I = 1 TO 3ØØ: NEXT I
17Ø GOTO 12Ø
```



7.
```
1ØØ REM ** SCROLL LEFT TO RIGHT **
11Ø CLS
12Ø A$ = " !WOW!     "
```

one space — 5 spaces

```
2ØØ REM ** DISPLAY SECTION **
21Ø I = LEN(A$)
22Ø PRINT @48Ø, MID$(A$,I,6);
23Ø FOR J= 1 TO 5Ø: NEXT J
24Ø I = I -1
25Ø IF I= Ø THEN 21Ø ELSE 22Ø
```

8. We're sorry, but that information is personal. Even Billy the Goat is entitled to some privacy.

CHAPTER ELEVEN

Arithmetic Functions

You have probably used two arithmetic functions — RND and INT — in some of your earlier programs. These two functions were discussed in the first book in this series and they were used in several programs in the current book. RND generates sequences of "random" numbers. INT operates on a number, drops the fractional part, and returns with the closest integer value that is less than or equal to the number that you gave to the function. For example:

You type:	The screen shows:
`PRINT RND(6)`	5
`PRINT INT(2.59)`	2
`PRINT INT(-1.7)`	-2

Try these examples on your TRS-80. Note: since RND is *random*, your screen can show any number from 1 through 6.

Your TRS-80 has many more functions that can operate on the numerical variables and constants within a program. All these functions are called *arithmetic* functions since they perform a variety of mathematical and computational tasks, as well as help you work with numbers within the TRS-80. All these functions return numeric values to the program.

Conversion Functions

Several functions exist to convert between integer, single precision, and double precision values. As a review, enter and RUN the next program that displays the same input values with different precisions:

Remember:
is double precision;
! is single precision;
% is integer.

```
100 REM ** PRECISION DISPLAY **
110 A# = 3.141592653589
120 A! = A#
130 A% = A#
140 CLS
150 PRINT "A# =";A#
160 PRINT "A! =";A!
170 PRINT "A% =";A%
```

Your screen should show this display when the program is RUN:

You've seen this before

```
A# = 3.141592653589
A! = 3.14159
A% = 3
READY
>-
```

Remember, these symbols (#, !, %), tell the TRS-80 that the variables are either double precision, single precision, or integer valued.

Now, with the screen showing the results of running the program, type the following line (*don't* clear the screen or type NEW):

```
PRINT 1/CDBL(A%)
```

What's this?

Does the screen show this?

```
A# = 3.141592653589
A! = 3.14159
A% = 3
>PRINT 1/CDBL(A%)
 .3333333333333333
READY
>-
```

Double precision result!!

Hmmm . . .

CDBL converted A% (an integer variable) to *double* precision thus forcing the entire expression to have a double precision result. This feature is handy to temporarily make a double precision computation with constants and variables that are either integers or single precision. In this way you don't have to put the values to be computed into a double precision variable location and do the arithmetic. You, therefore, save memory (remember how much room double precision variables occupy) and still can have the benefits of double precision computations. How would you convert from double to *single* precision? The typography gives you the answer — use CSNG.

Try it with the results of the current program by typing:

```
PRINT CSNG(A#)
```
← Try it!

A# is a double precision variable that contains the number 3.141592653589, once the program is RUN. What does your screen show when you enter the PRINT statement?

Is this what you now have?

Single precision result →

```
A# = 3.141592653589
A! = 3.14159
A% = 3
>PRINT 1/CDBL(A%)
 .3333333333333333
READY
>PRINT CSNG(A#)
 3.14159
READY
>-
```

CSNG performed the same operation as the program assignment of A! = A#. In each case, the value in A# was converted to a single precision number.

Have you guessed what the last function is? Yes, it is one that *con*verts values to *int*egers. The name of this function is CINT.

Type this line into your TRS-80, and see what happens:

```
PRINT CINT(A#)
```
← Try it also

The screen should now show:

Integer result →

```
A# = 3.141592653589
A! = 3.14159
A% = 3
READY
>PRINT 1/CDBL(A%)
 .3333333333333333
READY
>PRINT CSNG(A#)
 3.14159
READY
>PRINT CINT(A#)
 3
READY
>-
```

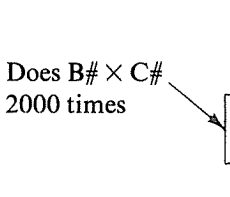
Using CINT results in the same final value as the program assignment A% = A#. The integer portion of the value is stripped from the number, with the fractional part dropped. CINT works the same way as INT. They both return an integer value that is the closest value less than the number or expression within the argument field. CINT is different in one respect — it only deals with values that are in the range of −32768 to +32767. Numbers outside that range produce an ?OV ERROR (overflow).

Try these examples on your TRS-80. What will be displayed?

```
PRINT CINT(2.75)  <------ 2 displayed
PRINT CINT(-1.4)  <------ -2 displayed
PRINT CINT(111111)  <---- ?OV ERROR displayed
```

Using the Conversion Routines

Significant time and memory savings are made by using the conversion routines. First enter and RUN the next program:




```
100 REM ** TIME TRIALS **
110 CLS
120 B# = 16/CDBL(3)
130 C# = 20/CDBL(3)
140 FOR I = 1 TO 2000
150   A = B# * C#
160 NEXT I
170 PRINT "CALCULATIONS COMPLETE"
```

The program takes a l-o-n-g time to run. The double precision variables B# and C# make the TRS-80 work to perform the multiplication shown in line 150. Time the program. The routine should make the 2000 calculations in about sixty seconds.

Now, suppose all you need at line 150 is the single precision portions of B# and C# to use in the computation. Make that change and see how long the program takes.

Reenter line 150 as follows:

```
150 A = CSNG(B#) * CSNG(C#)
```

Now RUN and time this program. Surprise! The program now executes in twenty-five seconds. By changing line 150, you told the TRS-80 to work with just the single precision portions of B# and C#. The result is a savings of thirty-five seconds when the program RUNs.

You must be anxious to try the last test — changing the calculation to integers. Go ahead, and enter this line:

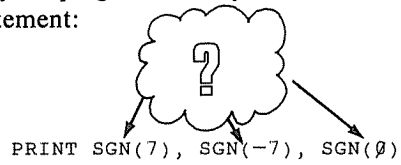
```
150 A% = CINT(B#)×CINT(C#)
```

What happens when you try this version of the program? Almost no change? That seems to be true; the program may run one or two seconds faster than the last version, but it is difficult to tell the difference. Most of the savings are made in just dropping the double precision, no matter how the drop occurs. You may want to experiment further with programs that test how fast the TRS-80 works. Go ahead and explore. Try changing the mathematical operation to addition, subtraction, division, and exponentiation. Vary the conversion functions and record all the times of execution. Learn how your TRS-80 works and you will enjoy its capabilities even more.

What Else Can You Do with Numbers?

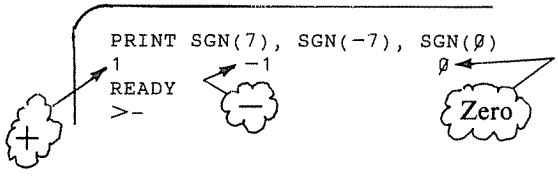
Type NEW and clear any old programs from your machine's memory.

Now enter this statement:



Look at the result on the screen. Can you guess what SGN is all about? Yes, it stands for the *SiGN* of the number; that is, SGN tells you whether a number is positive, negative, or zero. How?

Look again at the screen:



SGN returns a +1 when the number is positive; −1 when the number is negative; 0 when the number is 0. How can this function be used?

Enter the Oracle Program given below. This program tells you whether a number you enter is negative, zero, or positive.

Have you seen this before?

SGN(N)+2 is either 1,2, or 3

```
1ØØ REM "ORACLE PROGRAM"
11Ø CLS
12Ø INPUT "PLEASE, ENTER A NUMBER"; N

13Ø ON SGN(N)+2 GOTO 14Ø, 15Ø, 16Ø

14Ø PRINT "THE NUMBER IS NEGATIVE.": GOTO 12Ø
15Ø PRINT "THE NUMBER IS ZERO.": GOTO 12Ø
16Ø PRINT "THE NUMBER IS POSITIVE.":GOTO 12Ø
```

RUN the program and enter the three numbers 7, −7, and 0. The screen should show:

The oracle works

```
PLEASE, ENTER A NUMBER? 7
THE NUMBER IS POSITIVE.
PLEASE, ENTER A NUMBER? −7
THE NUMBER IS NEGATIVE.
PLEASE, ENTER A NUMBER? Ø
THE NUMBER IS ZERO.
PLEASE, ENTER A NUMBER? -
```

Try other values. Can you fool the oracle? Try entering −0 or +0. Does the program get those right? Yep, it's tough to fool an oracle. How about small numbers? Try .00000001 or −.000000000001. They work also!! Well, you might try:

.0000000000000000000000000000000000000001

40 zeros

What did you get? Aha, that number fooled the oracle. Actually, what happened is that the number is beyond the precision of the TRS-80 computer. The INPUT routine treats the number as it would a zero. The shorthand notation for the number is

1E-41

A 1 with 40 zeros to the left and the decimal point on the left end

So, the oracle wasn't fooled. The INPUT routine converted the number to a zero and then went on to the SGN test. Oh, well, the number 1E-41 is *almost* zero.

Try 1E-38 at the INPUT request. The program says that number is positive. Try 1E-39. What happens in this case? Yes, the program says that number is zero. Somewhere between 1E-38 and 1E-39 is the smallest number that the TRS-80 can accept as an INPUT. Maybe you can locate that number. Try .99E-38, then .98E−38, then . . ., then .95E-38, then .94E-38. You should see positive, positive, positive, and negative results, respectively.

Now you know it is somewhere between .95E-38 and .94E-38; you are narrowing in on it. The last number you tested was .9404E-38 and it tested positive. You might want to continue this search on your own.

When you get tired of looking for the smallest number, enter this next statement:

```
PRINT ABS(7), ABS(-7), ABS(Ø)
```

Another new function

Does the screen look like this?

```
PRINT ABS(7), ABS(-7), ABS(Ø)
7              7              Ø
READY
>-
```

Where is the minus sign?

You have probably guessed what ABS does. ABS *eats* minus signs; that is to say, ABS returns the *ABSolute* value of the argument it receives. If the argument is positive or zero, ABS of the argument is equal to the argument. If the argument is negative (arguments are almost always negative . . . discussions can be positive), ABS returns the argument without its minus sign. ABS negates negative arguments, making them positive. Is all of this clear? If not, just remember ABS *eats* minus signs.

ABS is best used to test the difference of two values, such as:

```
IF ABS(X2-X1)<.ØØØØ1 THEN STOP
```

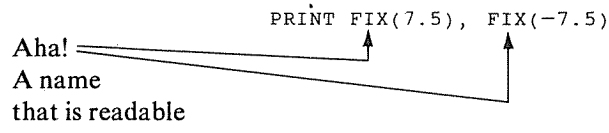
The statement shown causes the program to stop when the *absolute value* of the difference between the variables X2 and X1 is less than .00001 in value. The program stops even if X1 is bigger than X2, as long as the difference is between

$$(-.00001 < \textit{difference} < .00001$$

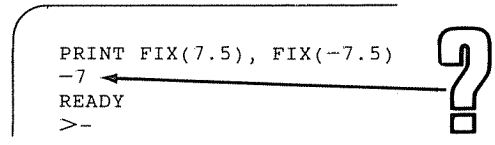
Without a function like ABS, the number of lines needed to make the test and stop the program increases, and the logic of making the tests gets complicated.

Let's examine one final function that tells you something about a number in your TRS-80. SGN tells you what the sign of a number is; ABS tells you what a number is, without its sign.

Enter this statement and determine what the new function tells you about a number:



Does your screen show:

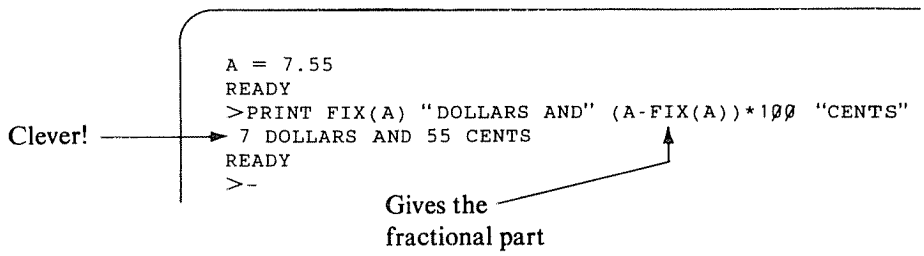


Hmmm . . . FIX looks a bit like INT or CINT. But, wait! There is one difference: FIX(−7.5) gives back a −7. INT and CINT would return a value of −8. FIX simply returns the integer portion of the number and strips away the fractional part. How can FIX be used?

Try this set of entries:

```
A = 7.55
PRINT FIX(A) "DOLLARS AND" (A-FIX(A))*1ØØ"CENTS"
```

Here is what the screen should show after making the two entries:



Anytime you need only the integer portion of the number, FIX can give you that value. Try a few FIXes in your own programs.

The Square Root Function

Up to this point, all the functions you have used dealt with manipulating numerical values (changing precision, locating the sign of the number, returning the absolute value, etc.). With the next function, you will perform your first true mathematical operation on a number.

Enter this statement:

In math notation, SQR(4) is $\sqrt{4}$ or $4^{1/2}$

```
PRINT SQR(4)
```

Back to funny names again

Did you get the number 2 printed on your screen? The number that was returned is the *square root* of the argument 4. In other words, 2 is the number that when squared (multiplied by itself — 2×2), gives the number 4. (Yep, $2 \times 2 = 4$.)

Now try:

```
PRINT SQR(2)      <-- √2
    and
PRINT SQR(121)    <-- √121
```

Did you get 1.41421 for SQR(2) and 11 for SQR(121)? Yes? Great! Let's see . . .
11×11 is easy to check. That multiplication gives 121 as an answer. But, what about 1.41421×1.41421? How can you check that one?

Use the TRS-80. Just type:

```
PRINT 1.41421*1.41421
```

What did you get? Oh! You got 1.99999 instead of 2. It is close, but not exact. What happened? Well, the square root of 2 is a number like some earlier ones you worked with that have endless fractional parts that never repeat. So, 1.41421 is only part of the number. Many more digits go after those shown. A still better way to check is to let the TRS-80 keep as many digits as it can while making the computation.

Try this version:

```
PRINT SQR(2)*SQR(2)
```

The answer comes back exactly 2 in this case. Is there any other way to check the answers? After all, using the function that produces the answers to check itself doesn't seem fair. Here is a short program that computes the square root using an *iterative* procedure (a routine that keeps making successive approximations of the answer and stops when the result is sufficiently close to the desired value).

Any positive value will work

Here is use of ABS

Make the old guess the new guess

```
10Ø REM ** NEWTON-RAPHSON EQUATION **
11Ø REM ** TO COMPUTE SQUARE ROOTS **
12Ø CLS

13Ø REM ** SET X1 TO ANY POSITIVE VALUE **
14Ø X1 = 5Ø

15Ø REM ** N IS VALUE FOR SQR(N) **
16Ø INPUT "ENTER VALUE "; N

17Ø REM ** COMPUTE APPROXIMATION **
18Ø X2 = .5*(X1 + N/X1)

19Ø REM ** CHECK FOR STOPPING **
2ØØ IF ABS(X2-X1) <.ØØØØØ1 THEN 24Ø

21Ø REM ** EXCHANGE X1 and X2 **
22Ø X1 = X2
23Ø GOTO 18Ø

24Ø REM ** DISPLAY RESULTS **
25Ø PRINT "THE SQUARE ROOT IS" X2
26Ø PRINT "THE SQR FUNCTION GIVES" SQR(N)
27Ø GOTO 16Ø
```

Enter and RUN this program using the values 4, 2, and 1E17 at the INPUT request. The screen should show:

The answers are the same

```
ENTER VALUE? 4
THE SQUARE ROOT IS 2
THE SQR FUNCTION GIVES 2
ENTER VALUE? 2
THE SQUARE ROOT IS 1.41421
THE SQR FUNCTION GIVES 1.41421
ENTER VALUE? 1E17
THE SQUARE ROOT IS 3.16228E+Ø8
THE SQR FUNCTION GIVES 3.16228E+Ø8
ENTER VALUE?
```

The routine seems to work, and both the iterative formula and the SQR function give the same answers. Do you suppose that the TRS-80 itself might be using the Newton-Raphson formula to compute square roots? So much for a *fair* test!

The program will not produce correct results for numbers whose roots are smaller than .000001 in value. Can you see why? Yes, the cutoff test uses that value to stop the iteration. To have the program get the correct roots of smaller numbers, change the value in line 200 to an appropriately small number. *Caution*: As the number is made smaller, the program runs for a longer time. Set up some time trials and test this fact yourself.

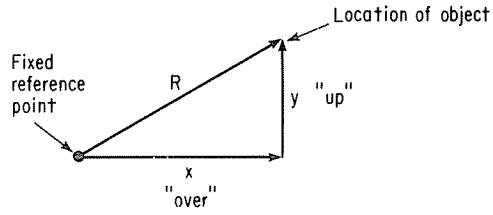
The SINE of the Times

To fully understand the TRS-80's *trigonometric* functions, you need a bit of geometry. Don't worry; you aren't expected to become a mathematician. The concepts are quite simple and are introduced only so you can use all of your computer's capabilities.

There are at least two ways to locate an object from a fixed position of reference:

(1) In terms of *coordinates* that measure how far over the object is and how far up.
(2) The distance to the object in a straight line, and an *angle* from some reference direction.

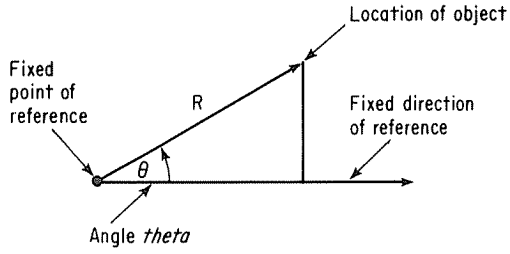
The first system looks like this:



This coordinate system is often called the Cartesian system after the French mathematician, René Descartes. The distance to the right is often labeled X; the distance up is labeled Y. The distance R (not D, since this distance relates to the *radius* of a circle . . . but that's for later) is given by the BASIC formula:

In math notation $R = \sqrt{x^2+y^2}$ ⟶ `R = SQR(X↑2 + Y↑2)`

The second system is called the *polar* coordinate system and is pictured in this way:

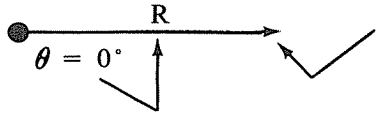


The two systems are related in that they both locate the object in the same place. The way they are related is through a set of functions called *trigonometric* functions. Trigonometric, or trig, functions are relationships between R, the straightline distance to the object, the X and Y of the Cartesian system, and the angle θ *(theta)* of the polar system. One function is called the *sine* of theta and is given by the equation:

$$\text{Sine}(\theta) = Y/R$$

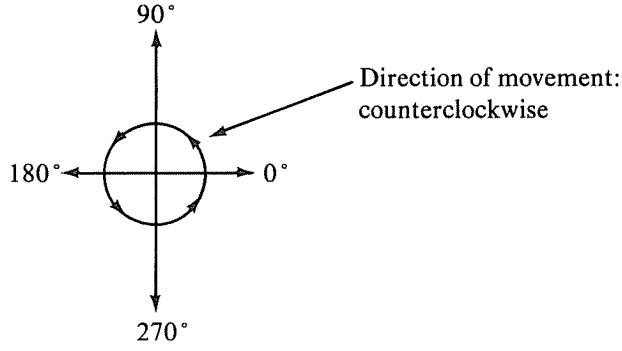
Now, the sine of the angle changes value as the angle changes. When theta is zero, Y is zero and the sine of theta is zero.

R is along reference coordinate



° Indicates degrees of angle

When theta is 90°, Y is equal to R and the sine of theta is 1. O.K. so far? Oh, yes, Theta goes from 0° along the reference direction to 90° when the object is directly above the fixed reference point, to 180° when the object is directly to the left, to 270° when the object is directly below, to 360° (or back to 0°) when the object is back at the starting position along the reference direction, as shown below:



How does the sine function behave as theta moves through all of the quadrants? It goes from 0 to 1 in the first quadrant; from 1 to 0 in the second; from 0 to −1 in the third; from −1 to 0 in the fourth. You can prove that these conditions hold by computing values for Y/R in all quadrants. A better way is to use the BASIC language function SIN (not evil, just short for *sine*). To do so requires you to learn one tiny detail. Most computers that have trig functions require that the angle theta be represented in a measurement called *radians*. Convert radians to degrees like this:

$$1 \text{ radian} = 57.29578^\circ$$

Magically enough, 2π radians is equal to $2 \times 3.14159 \times 57.29578$, or 360°.

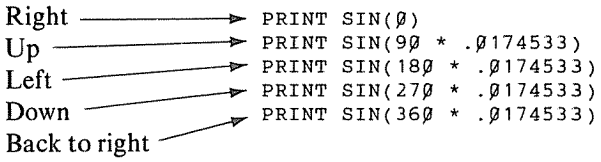
$$\boxed{2\pi = 360^\circ}$$

So, if 2π radians equals 360°, then the conversion from degrees back to radians is:

$$1 \text{ degree} = 2 \times 3.14159/360 = .0174533 \text{ radians}$$

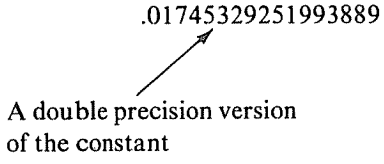
This last item is the punch line. In BASIC, to convert from degrees to radians, you multiply the angle (in degrees) by .0174533. If none of this makes sense, don't worry about it too much. Just use the SIN function with the constant that was developed and see if the function behaves as predicted. That result is the important part. Will the SIN function perform according to plan?

Try a few experiments and see what values SIN produces:



What did you get for each one of these PRINT statements? Did you get 0, 1, 0, −1, and 0? No!! The SINs of 180° and 360° came out close to zero but were off just a bit. Another error caused by precision.

Try them again with this constant:



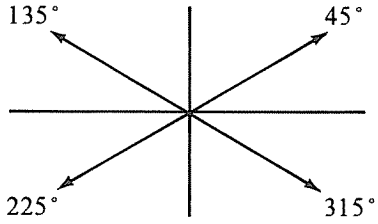
Did that constant fix things? Yes, it did. For the most part, you can stick with the shorter conversion factor. If you want to be *precise*, use the double precision number.

How about trying a few angles in between those already displayed?

Where are these on the diagram?

```
PRINT SIN(45 * .Ø174533)
PRINT SIN(135 * .Ø174533)
PRINT SIN(225 * .Ø174533)
PRINT SIN(315 * .Ø174533)
```

With the shorter constant the displayed values all look like either .70710Z or −70710Z, where the last digit Z is either 5, 6, 7, or 8. The variation is again caused by the loss of precision as related to the constant being used.



An Encouraging Sine

Trying a bunch of values with the SIN function tells you something about how it works. Of course, you could write a small program and have the SIN function plaster the screen with a lot of numbers. That exercise would tell you a bit more, perhaps. An even better way to examine the function, however, is to plot the results. Here is a small program that plots the SIN function on the TRS-80 display:

Makes a nice border for the plot

Aren't we precise!

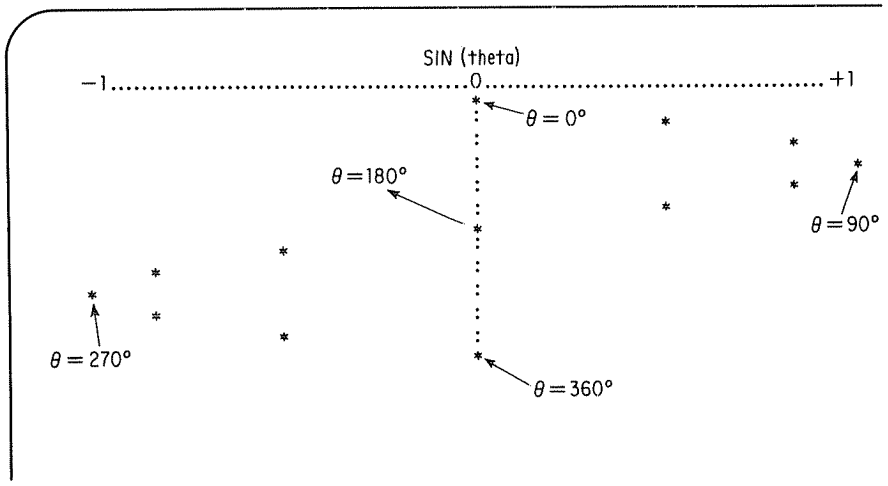
Plot

Wait

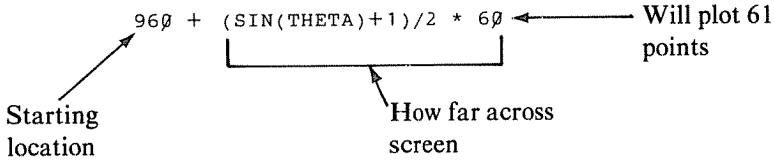
What's all this?

```
1ØØ REM ** SIN PLOT **
11Ø CLS
12Ø REM ** PUT HEADER ON PLOT **
13Ø PRINT @985,"SIN(THETA)"
14Ø PRINT @96Ø,"- 1  " STRINGS$(26,".") " Ø "
    STRINGS$(26,".") " +1"
15Ø REM ** BEGIN LOOP THRU 36Ø DEGREES **
16Ø FOR X = Ø TO 36Ø STEP 3Ø
17Ø   REM ** THETA IN RADIANS **
18Ø   THETA = X * .Ø1745329251993889
19Ø   REM ** PLOT CENTER LINE **
2ØØ   PRINT @99Ø,":";
21Ø   REM ** PLOT POINTS **
22Ø   PRINT @96Ø+(SIN(THETA)+1)/2*6Ø, "*"
23Ø NEXT X
24Ø GOTO 24Ø
```

Enter and RUN the program. The screen should show a plot of SIN(theta):



Line 220 is the key to this plotting routine. The expression in line 220:

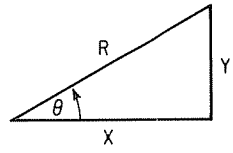


is used to position the plot symbol (*) across the screen.

Since SIN(theta) varies from −1 to +1, SIN(theta)+1 varies from 0 to 2. Dividing this expression by 2 gives a range of values from 0 to 1. Multiplying by 60 and adding 960 sets up values from 960 to 1020. These last values represent 61 print positions across the bottom of the screen. As each line is PRINTed, the screen scrolls upward to make room for the next line.

Back to the Geometry Lesson

Now that you have the SIN program working, you can easily look at one other trig function. Referring back to the diagram:



there is a co-relationship of theta, R, and X, called the *cosine*. The cosine is given by the equation:

$$\text{COSINE}(\theta) = X/R$$

Can you estimate how this function behaves? Why bother! Just make a few adjustments to the SIN program and let the TRS-80 do it for you.

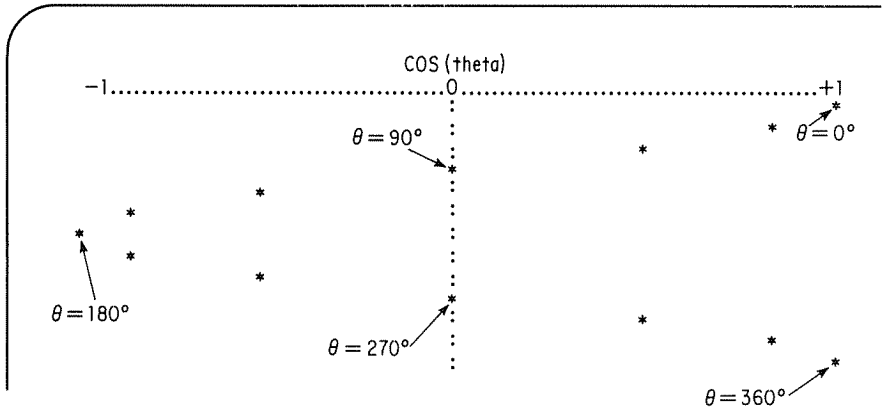
Here goes; make the following changes:

Change the program in these two lines →

```
130 PRINT @985, "COS(THETA)"
220 PRINT @960+(COS(THETA)+1)/2*60, "*"
```

This expression assumes that the COS will be in the range of −1 to +1

What appears when you RUN this program? Does your screen show:



The COS produces the same values as SIN, but everything is /"90 out of phase." When the SIN is zero, the COS is either +1 or −1. When the SIN is either +1 or −1, the COS is zero. In fact, it can be proven (but we won't attempt to do so here) that:

$$\text{SIN}(\theta)^2 + \text{COS}(\theta)^2 = 1$$

You may want to write a program that checks out the last statement. Your turn to experiment. Try some other operations with SIN and COS.

Off on a TANgent

One obvious relationship remains to be explored in our small geometry problem. Since the SIN is determined by Y/R and the COS is determined by X/R, there is a third expression that doesn't involve R. It can be obtained by dividing SIN by the COS. Here is the result:

$$SIN(\theta)/COS(\theta) = Y/R \ / \ (X/R) = Y/X$$

This function, called the *tangent*, is referred to in BASIC as TAN. The following program tabulates the trig functions from 0 to 80 in steps of ten.

Program to list trig functions

```
100 REM ** TRIG FUNCTIONS **
110 CLS

120 REM ** HEADER **
130 PRINT "SIN","COS","TAN","SIN/COS"
140 PRINT "---","---","---","-------"

150 REM ** BEGIN LOOP THRU ANGLES **
160 FOR X = 0 TO 80 STEP 10

170   REM ** COMPUTE THETA **
180   TH = X * .0174532925199 3889

190   REM ** DISPLAY RESULTS **
200   PRINT SIN(TH),COS(TH),TAN(TH),
      SIN(TH)/COS(TH)

210 NEXT X
```

Can you think of a reason for the program not looping to 90°? Try to think of what would happen, as you enter, RUN, and watch the current program execute.

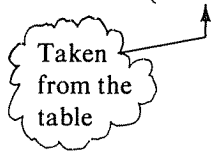
Your screen should show:

```
 SIN          COS          TAN          SIN/COS
 0            1            0            0
 .173648      .984808      .176327      .176327
 .34202       .939693      .36397       .36397
 .5           .866025      .57735       .57735
 .642788      .766044      .8391        .8391
 .766044      .642788      1.19175      1.19175
 .866025      .5           1.73025      1.73025
 .939693      .34202       2.74748      2.74748
 .984808      .173648      5.67129      5.67129
READY
>-
```

Ahh! Symmetry and order! Notice how the SIN and COS have the same values, but in reverse order. Additionally, SIN/COS *is* the same as TAN. Maybe that is how the TRS-80 generates TAN. The two columns *are* remarkably the same. Have you guessed what would happen if the loop continued to 90°? The TAN function is growing quite rapidly. Yes, TAN goes to infinity at 90°! Why? Look at the COS. At 90°, the COS is zero, while the SIN has the value one. One divided by zero is . . . a *big* number! Could it be that this is where the phrase "off on a tangent" got started? Seems a likely coincidence, doesn't it?

The TRS-80 also provides an *inverse* trig function called the *arctangent*, ATN. ATN returns a value in radians. To get back to degrees, multiply the result by 57.29578. For example:

```
PRINT ATN(5.67129) × 57.29578
```



What do you get when you enter the PRINT statement? Do you get a result of 80°? The TAN of 80° was the last entry in the table on the last page. The TAN of 80° is 5.67129. The arctangent (inverse tangent) of 5.67129 is, of course, 80°. Ahhh . . . the world *is* full of order and symmetry.

With these four trig functions — SIN, COS, TAN, and ATN — you can construct many other trig functions. Appendix F of your *Level II BASIC Reference Manual* lists several other trig functions and how to derive them from the four basic functions. You can even construct the Inverse Hyperbolic Cosecant with a little help from the next two mathematical functions.

What is the AnaLOG of "EXPect Little?"

Another constant important in mathematics is called *e*. It is an endless number that looks like this:

$$e = 2.7182818284590452\ldots$$

TRS-80 BASIC provides a function that allows you to compute powers of *e* easily. The function is called EXP, and EXP(X) represents raising *e* to the X power:

EXP(X) represents e^X

To examine the behaviour of EXP, enter and RUN this program:

```
10Ø REM ** EXPLORING EXP **
11Ø CLS

12Ø FOR X = Ø TO 1Ø
13Ø   PRINT EXP(X/1Ø)
14Ø NEXT X
```

A RUN of the program should produce this listing on the display:

```
 1                    <-- e^0
 1.1Ø517
 1.2214
 1.34986
 1.49182
 1.64872
 1.82212
 2.Ø1375
 2.22554
 2.4596
 2.71828              <-- e^1
READY
>-
```

EXP is exponential in nature. As the argument gets larger, EXP result gets larger as well. In fact, EXP gets big fast. If EXP has an argument greater than 87, an ?OV ERROR will occur.

Change line 120 and experiment with other values of EXP. Try some negative values. What happens when you put negative values into EXP? Try some *large* negative values. Think about how you could plot EXP on the screen.

EXP also has an inverse function, the *natural logarithm*. Let's modify the current program to display LOG values. Make these changes:

```
12Ø FOR X = 1 TO 1Ø          <-- Note range change
13Ø PRINT LOG(X/1Ø)
```

RUN this program and observe the numbers on the screen:

```
-2.3Ø259             <-- Log_e(.1)
-1.6Ø944
-1.2Ø397
-.916291
-.693147
-.51Ø826
-.356675
-.223143
-.1Ø536
 Ø                   <-- Log_e(1)
READY
>-
```

The values of the arguments to LOG must be positive (greater than zero). At zero, LOG goes to *minus* infinity (a large negative number). At one, LOG(1) equals zero. As the argument of LOG gets larger, LOG grows, but slowly.

Once again, try some values for LOG on your own. Look at those cases where the arguments get big (1E38). Think how you might go about plotting LOG on the screen. LOG and EXP are inverse functions. Each is the inverse of the other. Try to prove this by entering PRINT EXP(LOG(10)). What happens? Now try PRINT LOG(EXP(10)). What happens with this one? Did you get the same answer in each case?

Summary

You have been exploring the TRS-80 arithmetic functions. These functions assist in performing mathematical calculations and doing mathematically based studies. The functions you have examined are:

- CDBL(X) — Converts X to double precision
- CSNG(X) — Converts X to single precision
- CINT(X) — Converts X to an integer (− 32768 to +32767)
- SGN(X) — Determines the sign of X
- ABS(X) — Computes the absolute value of X
- FIX(X) — Truncates the fractional part of X
- SQR(X) — Computes the square root of X
- SIN(X) — Computes the trigonometric sine of X. X must be in radians. If X is in degrees, multiply X by .0174533
- COS(X) — Computes the trigonometric cosine of X. X must be in radians
- TAN(X) — Computes the trigonometric tangent of X. X must be in radians
- ATN(X) — Computes the inverse tangent of X. Answer is in radians. To convert to degrees, multiply by 57.29578
- EXP(X) — Computes the exponential e^x
- LOG(X) — Computes the natural logarithm of X, $log_e(X)$

You also saw how you might plot some function, such as SIN or COS, on the TRS-80 screen by using the function to calculate where the plot points were to fall. Try your hand at the self-test that follows. Then move on to the last chapter in the book — a chapter of fun and games.

Self-Test

1. What are the names of the functions that convert numbers or expressions to:
   (a) Double precision _____ (b) Single precision _____
   (c) Integers _____
2. If the variable A# = 2.7182818282845, what would show on the screen when you type: `PRINT CSNG(A#)` (a) _____

   What appears when you type: `PRINT CINT(A#)` (b) _____
3. If the following line is in a program, how would you change it so the program executes faster?

```
12Ø A% = B# * C#
```

4. What appears on the screen when you type these statements into your TRS-80?
   (a) `PRINT SGN(15.5)`_____ (b) `PRINT SGN(-12)`_____
   (c) `PRINT SGN(Ø)` _____
5. What does this statement display on the screen? _____

```
PRINT ABS(-17.7)
```

6. How is the FIX function different from the INT function?

7. What does the TRS-80 give for:

```
PRINT SQR(16), 16↑(1/2)
```

8. What is the factor that converts angles in degrees to radians?

9. What is the factor that converts radians into degrees of angle?

10. What does the following PRINT statement produce on the screen:

```
PRINT EXP(LOG(2Ø)), LOG(EXP(2Ø))
```

Answers to Self-Test

1. (a) CDBL (b) CSNG (c) CINT
2. (a) 2.71828 (b) 2
3. One way: `12Ø A% = CINT(B#) * CINT(C#)` if the integer values are in the range of −32768 to 326767.

   Another way: `12Ø A% = CSNG(B#) * CSNG(C#)`
4. (a) 1 (b) −1 (c) 0
5. 17.7
6. FIX truncates the fractional part of the number and returns the integer portion. INT truncates the fractional part, but returns greatest integer less than the original value. So, for negative numbers FIX(−7.5) returns −7; INT (−7.5) returns −8.
7. Two numbers; both 4. SQR(16)=4, and 16↑(1/2) is an alternate way to compute the square root.
8. .0174533 or .01745329251993889 to be *precise*. Actually, even the last number is not precise; there is more. The factor goes on forever.
9. 57.29578—multiplying 2π times the number gives 360.
10. The screen shows the number 20 printed twice. The LOG and EXP functions are inverses of each other so the expressions in the PRINT statement return the argument 20.

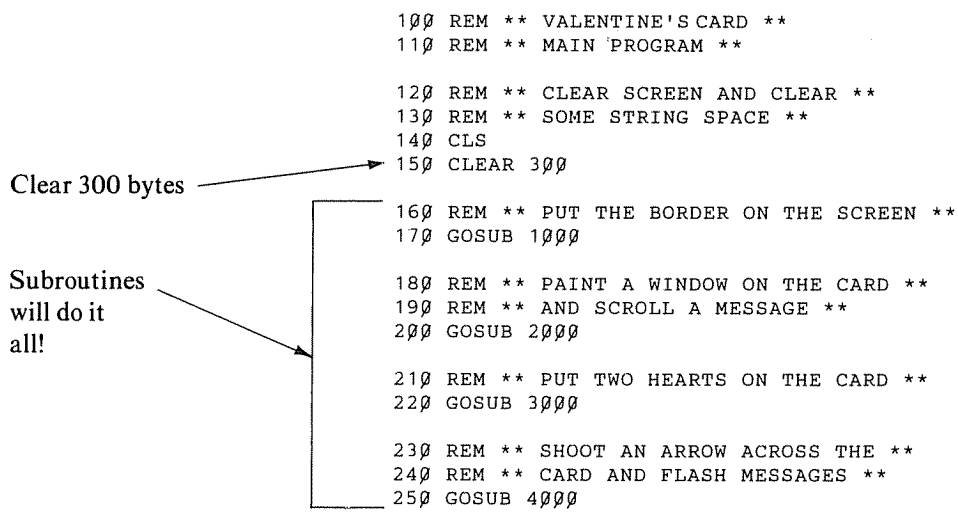
CHAPTER TWELVE

TRS-80 Art Lesson

You are nearly to the end of this book. Time to relax and use some of your TRS-80's features you have been reading about. Since the graphics on the TRS-80 are so easy to work with, let's begin by creating an animated Valentine's Day card for your mate or companion.

Electronic Love Notes

Your machine's graphic features can produce an animated version of a Valentine's Day card. Start by building the main program; a series of GOSUBs to routines to draw the card's border, scroll the message across the face of the card, draw some hearts, shoot arrows, and cause other messages to flash. (We bet no one has ever received a card like this before.) Here is the main program:




```
100 REM ** VALENTINE'S CARD **
110 REM ** MAIN PROGRAM **

120 REM ** CLEAR SCREEN AND CLEAR **
130 REM ** SOME STRING SPACE **
140 CLS
150 CLEAR 300

160 REM ** PUT THE BORDER ON THE SCREEN **
170 GOSUB 1000

180 REM ** PAINT A WINDOW ON THE CARD **
190 REM ** AND SCROLL A MESSAGE **
200 GOSUB 2000

210 REM ** PUT TWO HEARTS ON THE CARD **
220 GOSUB 3000

230 REM ** SHOOT AN ARROW ACROSS THE **
240 REM ** CARD AND FLASH MESSAGES **
250 GOSUB 4000
```

To close off the main part of the program, a delay routine is called and the program redisplays the card.

Wait a while . . .

```
26Ø REM ** WAIT AND THEN RE-DISPLAY **
27Ø T = 1ØØØ: GOSUB 5ØØØ
28Ø GOTO 14Ø
```

Now let's develop the routine to draw the border. The statements used are taken from several previous discussions in this book dealing with the rapid placement of graphic characters on the screen.

Solid bars across the top and bottom

Thin strips down the sides

```
1ØØØ REM ** OUTLINE THE SCREEN **
1Ø1Ø REM ** TOP AND BOTTOM FIRST **
1Ø2Ø PRINT @Ø,STRING$(64,191);
1Ø3Ø PRINT @896, STRING$(64,191);

1Ø4Ø REM ** NOW THE TWO SIDES **
1Ø5Ø FOR I = 64 TO 832 STEP 64
1Ø6Ø   PRINT@I,CHR$(191);        <- Left
1Ø7Ø   PRINT@I+63, CHR$(191);    <- Right
1Ø8Ø NEXT I

1Ø9Ø RETURN
```

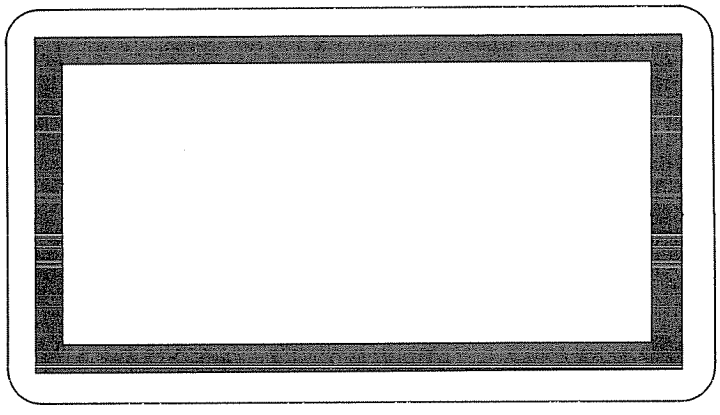
Check this portion of the program by entering "dummy" routines for the rest of the subprograms to be developed. Type in the delay routine directly since you know what it looks like:

"Dummy" routines

Count up to T

```
2ØØØ RETURN
3ØØØ RETURN
4ØØØ RETURN

5ØØØ REM ** DELAY ROUTINE **
5Ø1Ø FOR TT = 1 TO T
5Ø2Ø NEXT TT
5Ø3Ø RETURN
```

RUN this partially completed program. Your screen should show:



Now you are ready to enter the next routine — the one to open a window area on the card and scroll a message across the window.

Enter the following statements:

```
2000 REM ** WINDOW AND MESSAGE SCROLL **
2010 REM ** PAINT THE WINDOW **
2020 PRINT @267, STRING$(38,191);

2030 REM ** SET UP THE MESSAGE **
2040 A$= STRING$(37,191) +
     "...HAPPY VALENTINE'S DAY..." +
     STRING$(6,191)

2050 REM ** SCROLL THE MESSAGE TWICE **
2060 REM ** HORIZONTALLY ACROSS SCREEN **
2070 FOR K = 1 TO 2
2080   I = 1
2090   PRINT@267, MID$(A$,I,38);
2100   T = 25: GOSUB 5000
2110   I = I + 1
2120   IF I<= LEN(A$) THEN GOTO 2090
2130 NEXT K

2140 REM ** PUT MESSAGE IN CENTER **
2150 REM ** OF THE WINDOW **
2160 PRINT@267, MID$(A$,32);

2170 RETURN
```

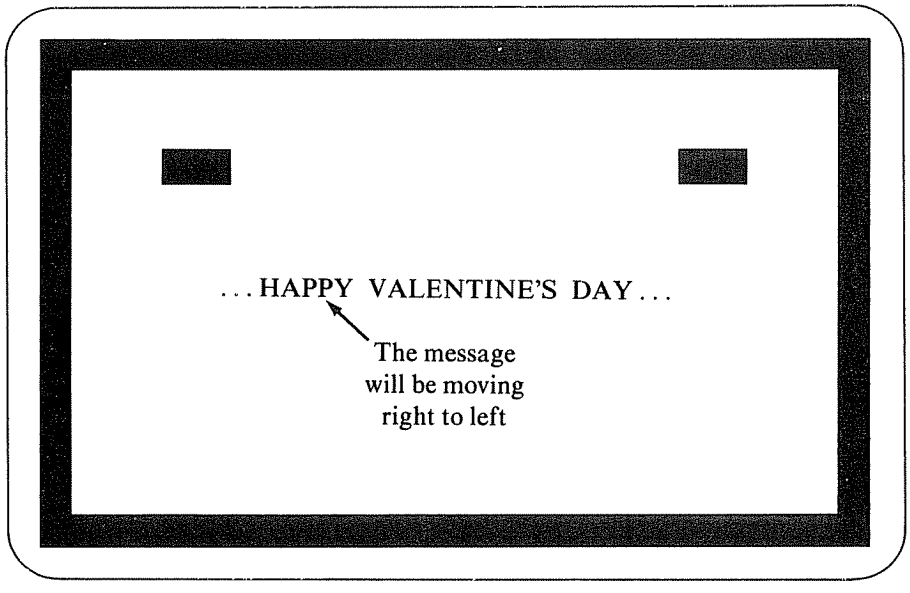
Window

37 , followed by message and six more

You did this in chapter 10

Begins with 32nd character and takes rest of message

Go ahead and RUN this much of the program. The screen should clear, the border should appear, then the window, and finally the Valentine's Day message should scroll from right to left on the face of the window.



Easy enough? You are simply combining several features used in previous chapters. Ready for the next part? O.K. how about the routine that draws the hearts?

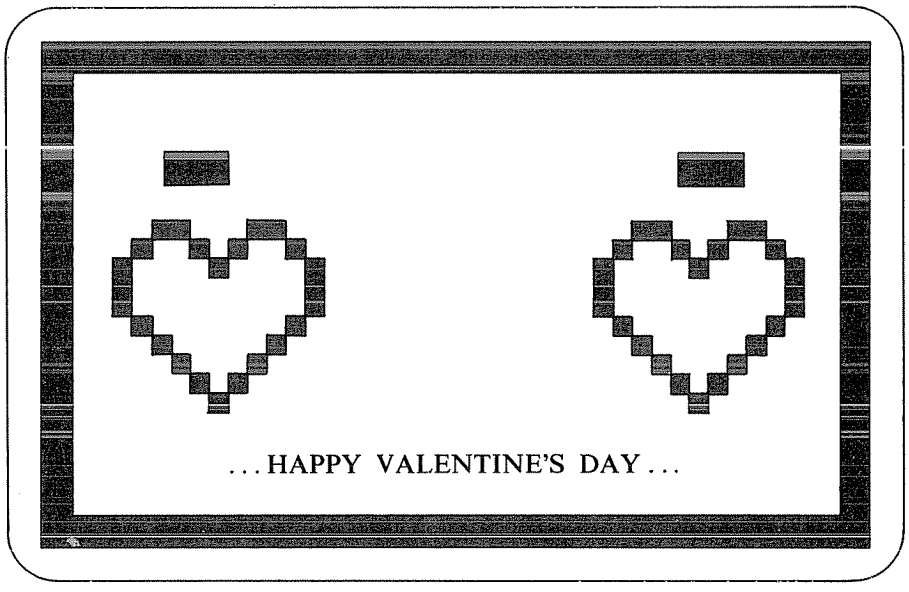
Enter these next statements and RUN the new program:

```
3000 REM ** DRAW TWO HEARTS **
3010 REM ** DATA VALUES FOR SPOTS **
3020 REM ** TO BE SET **
3030 DATA 21,22,19,21,17,20,15,20,13,20
3040 DATA 11,21,9,22,9,23,9,24,9,25
3050 DATA 11,26,13,27,15,28,17,29,19,30
3060 DATA 21,31,23,30,25,29,27,28,29,27
3070 DATA 31,26,33,25,33,24,33,23,33,22
3080 DATA 31,21,29,20,27,20,25,20,23,21
3090 DATA -99,-99

3100 REM ** READ THE PAIRS OF DATA **
3110 READ X,Y
3120 REM ** CHECK FOR END OF DATA **
3130 IF X= -99 THEN GOTO 3180
3140 REM ** SET TWO POINTS **
3150 SET(X+XINC,Y): SET(X+XINC-1,Y)
3160 GOTO 3110
3170 REM ** CHECK TO SEE IF COMPLETE **
3180 IF XINC<>0 THEN RETURN

3190 REM ** ONE HEART DRAWN **
3200 REM ** ADJUST XINC AND DRAW 2ND **
3210 XINC = 78
3220 RESTORE
3230 GOTO 3110
```

What is XINC used for?

Sets XINC so that X+XINC puts heart on right side of screen

Do the two hearts appear on the screen when you RUN the program? Doesn't that make your heart throb?



The DATA values in the last subroutine provide the locations of the various spots used to create one heart diagram. The heart on the left side of the card is drawn first. Then the variable XINC is changed from zero (0) to 78, the DATA statements are RESTOREd, and the second heart appears. When the second heart is complete, XINC is non-zero, and a RETURN is made to the main program.

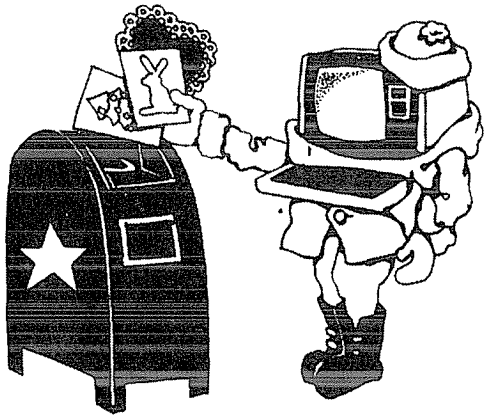
Are you ready for the grand finale? Yes!! Well, here goes! Enter the statements for the last subprogram and RUN the complete Valentine's Day Card program:

Animation of arrow

Takes care of printing and flashing the messages

```
4000 REM ** SHOOT AN ARROW **
4010 REM ** AND FLASH THE MESSAGES **
4020 REM ** THIS IS AN ARROW **
4030 B$ = "----->"

4040 REM ** SHOOT THE ARROW THREE TIMES **
4050 FOR K = 1 TO 3

4060   REM ** ARROW DISPLAY LOOP **
4070   FOR I = 0 TO 19
4080     PRINT@529,STRING$(I,32)+B$;

4090     REM ** TEST FOR FIRST TIME **
4100     REM ** THRU K LOOP **
4110     IF K = 1 THEN GOTO 4230

4120     REM ** DISPLAY LOVE MESSAGE **
4130     PRINT@724,"I L O V E Y O U";

4140     REM ** WHEN I IS GREATER THAN **
4150     REM ** TEN, FLASH THE LOVE NOTE **
4160     IF I>10 THEN T=50: GOSUB 5000
4170     IF I>10 THEN PRINT @274,
         STRING$(20,30);

4180     REM ** IF I IS GREATER THAN **
4190     REM ** TEN, AND THIS IS THE **
4200     REM ** THIRD ARROW SHOT, FLASH **
4210     REM ** THE TOP MESSAGE ALSO **
4220     IF I>10 AND K=3 THEN PRINT@267,
         STRING$(40,32);
4230     T = 50: GOSUB 5000
4240     IF I>10 AND K=3 THEN PRINT@267,
         MID$(A$,32);

4250   NEXT I

4260   REM ** BLANK OUT ARROW **
4270   PRINT@548,STRING$(6,32);

4280 NEXT K
4290 RETURN
```

The arrow flashes across the screen from heart to heart. When it hits the first time, the message I LOVE YOU appears near the bottom of the card. As the arrow gets halfway across on its second pass, the love message begins to flash. On the third pass of the arrow, both the love message and the message at the top of the screen begin to flash. Now, that's a Valentine's Card! If you have friends who own computers, you can make up cassettes during holiday seasons or on special occasions and send everyone a TRS-80 electronic greeting card. Enjoy using your TRS-80 in this way.



Oh, yes! The kinds of routines used in making this card can also be used in programs and games that you develop. The border display, the scrolling feature, the flashing messages, the animated arrow, and the drawing of the hearts indicate the many methods of display you can use on your computer. Think of other things you can do with these features. What kind of games and programs can *you* invent to use the TRS-80 this way?

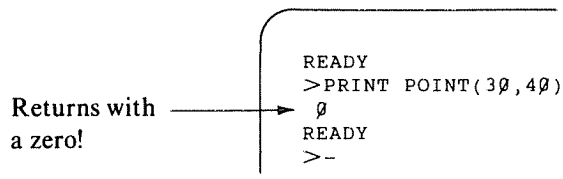
Let's Get to the POINT

Have you had a chance to use the POINT function yet? If not, try the following:

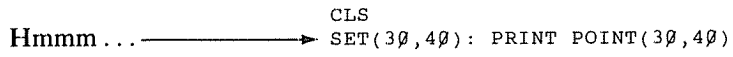
Where does the TRS-80 POINT to? ⟶

```
CLS
PRINT POINT(3Ø,4Ø)
```

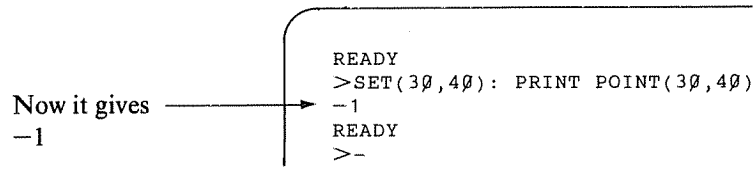
What now shows on your screen? Does it look like this?



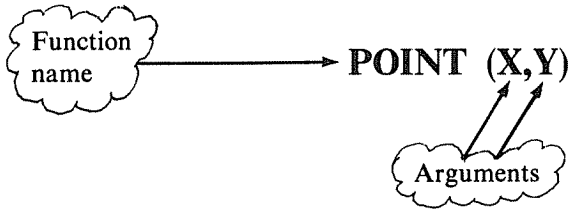
Now, enter these statements and observe the screen:



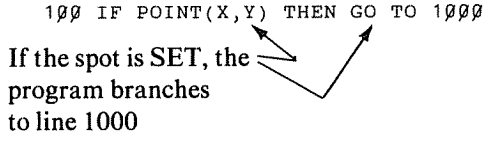
What does your screen show now? Is this what appears:



Can you guess what POINT is doing? The function "looks" at the place on the screen specified by the arguments:



If that spot on the screen is SET, then POINT returns the value −1. If the spot is not SET, POINT returns a value of zero (0). For the TRS-80, the values of −1 and 0 represent the logical values of "true" and "false." Minus one (−1) indicates that a condition is true; zero (0) indicates that a condition is not true, or false. This set of facts allows you to put the POINT function in an IF-THEN statement and perform some interesting feats.



As an example, let's develop a program that uses POINT to determine if the screen is SET and if the program is to RESET that spot. To get interesting results, use a random process for determining where the next spot is to be RESET. Hmmm . . . seems as if another TRS-80 first is about to happen in the computer world.

The Finer Art of POINTing

Here is the first part of the program:

```
100 REM ** TRS-80 ART **
110 REM ** CLEAR SCREEN AND STRING SPACE **
120 CLS
130 CLEAR 100

140 REM ** SET ON ERROR CONDITION **
150 ON ERROR GOTO 1000

160 REM ** PAINT THE SCREEN WHITE **
170 FOR I = 0 TO 896 STEP 64
180   PRINT@I, STRING$(64,191);
190 NEXT I

1000 RESUME
```

Remember this feature?

You've done this before

For the ON ERROR routine

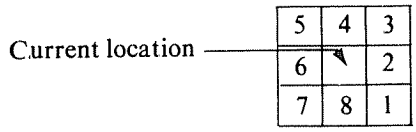
Nothing special up to this POINT (oops! . . . point). The ON ERROR condition takes care of a problem you will encounter later as the POINT routine tries to "look" off the edge of the screen.

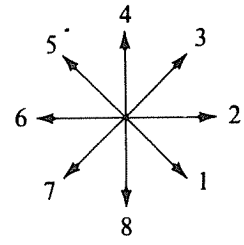
RUN this part. The screen should turn white, and then an error should occur (when the RESUME is encountered), but ON ERROR the program goes to 1000. That place in the program is causing the error, so the result is like putting 1000 GOTO 1000 in the program. Anyway, the screen turns white. What's next?

Now add this part of the program that performs the random RESET of the spots on the screen:

```
200 REM ** START NEAR THE MIDDLE **
210 X = 63: Y = 23
220 RESET(X,Y)

230 REM ** PICK A RANDOM DIRECTION **
240 REM ** TO POINT **
250 R = RND(8)
```

A given spot on the screen has eight surrounding spots (unless the current location is on the edge of the white area). The locations are numbered in this way. You could change this assignment; the numbering is arbitrary.





Possible directions

The next step is to POINT at the randomly chosen location, and if it is SET, to RESET it.

Enter the following:

```
26Ø REM ** SAVE THE CURRENT LOCATION **
27Ø XX = X: YY = Y

28Ø REM ** ADJUST X,Y TO POINT AT SPOT **
29Ø REM ** CHECK FOR SPOT ON RIGHT **
3ØØ IF R<4 THEN X = X + 1
31Ø REM ** CHECK FOR SPOT ON LEFT **
32Ø IF R>4 AND R<8 THEN X = X - 1
33Ø REM ** CHECK FOR UP **
34Ø IF R>2 AND R<6 THEN Y = Y - 1
35Ø REM ** CHECK FOR DOWN **
36Ø IF R>6 OR R=1 THEN Y= Y + 1
```

Takes care of right-left

Takes care of up-down

The adjustments to X and Y can produce values that are off the screen. The ON ERROR statement that was used in line 150 lets the program POINT off the screen and *not* abort the program.

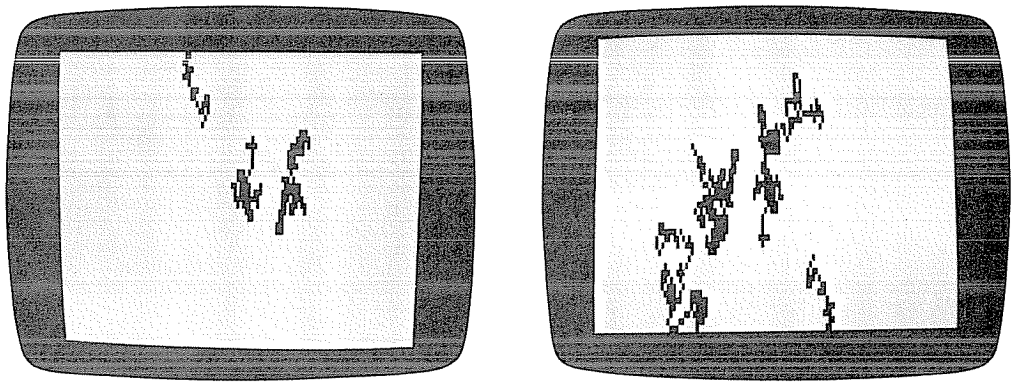
Here is the POINT you have been waiting for:

```
38Ø REM ** CHECK TO SEE IF SPOT IS SET **
39Ø IF POINT(X,Y) THEN GOTO 22Ø

4ØØ REM ** SPOT NOT SET, CHOOSE ANOTHER **
41Ø REM ** RESTORE X AND Y **
42Ø X = XX: Y = YY
43Ø GOTO 25Ø

1ØØØ RESUME 42Ø
```

Yes, we skipped line 370

Change line 1000 also

Run this program. The screen should turn white; then a series of small black holes should start to appear. The "pictures" sketched on the screen will probably never be the same because of the random way the spots are chosen and RESET make each "drawing" unique. Here are a few examples of designs we produced:



Notice that the POINTing process often causes the program to be at a location where all the surrounding spots are already RESET. The program then enters a loop from which it never exits: choosing a random spot, pointing at it, finding it RESET, and going back to choose another random spot. Hitting the BREAK key stops the program so you can re-RUN it to produce another sketch. Would you like to "move" the current spot when the program gets into a loop? Very well; try these additions.

Your Move, Leonardo

The only place a message can be placed on the screen and not disturb the drawing is on the bottom line. Using PRINT @ allows a message to be put there without causing the screen to scroll upward. To prevent scrolling upon INPUT, a clever use of INKEY$ has been devised.

Enter the following:

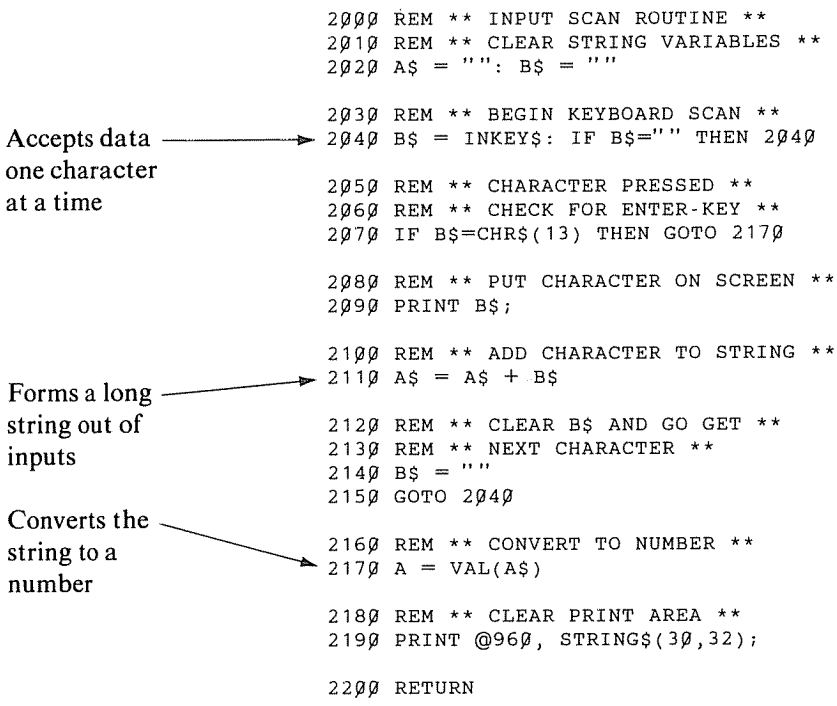
Here is line 370 ——→

Pressing any key will interrupt the program now

```
37Ø IF INKEY$<>"" THEN 5ØØ

5ØØ REM ** ACCEPT CHANGES TO X AND Y **
51Ø REM ** FIRST REQUEST X CHANGES **
52Ø PRINT @96Ø, "ENTER X'S MOVE(+ OR -)";

53Ø REM ** USE SUBROUTINE TO ACCEPT INPUT **
54Ø GOSUB 2ØØØ

55Ø REM ** INCREMENT IS IN A **
56Ø XX = XX + A

57Ø REM ** Y CHANGE REQUEST **
58Ø PRINT @96Ø, "ENTER Y'S MOVE(+ OR -)";
59Ø GOSUB 2ØØØ
6ØØ YY = YY + A
61Ø GOTO 42Ø
```

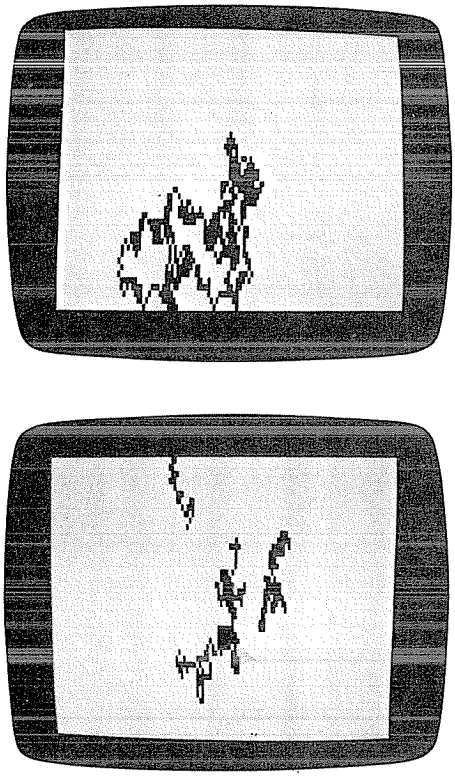
The subroutine at line 2000 takes care of accepting the INPUT to the model.

Here is that routine:




```
2000 REM ** INPUT SCAN ROUTINE **
2010 REM ** CLEAR STRING VARIABLES **
2020 A$ = "": B$ = ""

2030 REM ** BEGIN KEYBOARD SCAN **
2040 B$ = INKEY$: IF B$="" THEN 2040

2050 REM ** CHARACTER PRESSED **
2060 REM ** CHECK FOR ENTER-KEY **
2070 IF B$=CHR$(13) THEN GOTO 2170

2080 REM ** PUT CHARACTER ON SCREEN **
2090 PRINT B$;

2100 REM ** ADD CHARACTER TO STRING **
2110 A$ = A$ + B$

2120 REM ** CLEAR B$ AND GO GET **
2130 REM ** NEXT CHARACTER **
2140 B$ = ""
2150 GOTO 2040

2160 REM ** CONVERT TO NUMBER **
2170 A = VAL(A$)

2180 REM ** CLEAR PRINT AREA **
2190 PRINT @960, STRING$(30,32);

2200 RETURN
```

When the program POINTs itself into a corner, press any key. The X-change message appears at the bottom of the screen. As you press the number keys, they are displayed on the same line, but when you press ENTER, the screen does not scroll upward. The Y-change message appears next. You can enter either positive or negative increments to move the current spot. RUN this altered program. Here are some more examples of the "art" produced by this program:



You can tell the program to move the spot off the screen. If you do so, just press ENTER again and have it move it back. For example, let's say you are at the right edge and enter a 30 for the X= change. The X address is now 157 (127 + 30). The program just goes into a loop. Press ENTER, and at the X-change request type in −60. The X address is now 97 (157−60). Giving it a value for the Y-change (which can be zero), causes the program to resume sketching in the white area of the display. Try telling the program to sketch off the screen, and then tell it to move back. Try other experiments of your own.

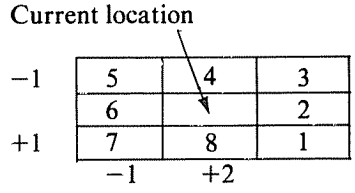
Experimental Art

You may want to try some other experiments with the TRS-80 Art Program. What happens in lines 300 through 360 when you change the increments (+1 and −1) by which X and Y move? For example, what do you suppose happens when the X increments are set to +2 and −2 in lines 300 and 320? Here is a picture of the spots that are RESET.

Current location

−1	5	4	3
	6		2
+1	7	8	1
	−2	+2	

What would occur if only line 300 is changed? Again, here is a diagram of what spots are RESET:



Try these experiments. You might want to alter the program so that the increments are INPUT to the program. Doing so gives you more freedom in trying sets of values to see how the program responds. Experiment away! For your convenience, the Valentine's Card and TRS-80 Art Programs are listed on these final pages. Enjoy using them and your TRS-80 microcomputer.

Valentine's Card Program

```
100 REM ** VALENTINE'S CARD **
110 REM ** MAIN PROGRAM **

120 REM ** CLEAR SCREEN AND CLEAR **
130 REM ** SOME STRING SPACE **
140 CLS
150 CLEAR 300

160 REM ** PUT THE BORDER ON THE SCREEN **
170 GOSUB 1000

180 REM ** PAINT A WINDOW ON THE CARD **
190 REM ** AND SCROLL A MESSAGE **
200 GOSUB 2000

210 REM ** PUT TWO HEARTS ON THE CARD **
220 GOSUM 3000

230 REM ** SHOOT AN ARROW ACROSS THE **
240 REM ** CARD AND FLASH MESSAGES **
250 GOSUB 4000

260 REM ** WAIT AND THEN RE-DISPLAY **
270 T = 1000: GOSUB 5000
280 GOTO 140

1000 REM ** OUTLINE THE SCREEN **
1010 REM ** TOP AND BOTTOM FIRST **
1020 PRINT @0,STRING$(64,191);
1030 PRINT @896, STRING$(64,191);

1040 REM ** NOW THE TWO SIDES **
1050 FOR I = 64 TO 832 STEP 64
1060   PRINT@I, CHR$(191);
1070   PRINT@I+63, CHR$(191);
1080 NEXT I

1090 RETURN
```

```
2000 REM ** WINDOW AND MESSAGE SCROLL **
2010 REM ** PAINT THE WINDOW **
2020 PRINT @267, STRING$(38,191);

2030 REM ** SET UP THE MESSAGE **
2040 A$ = STRING$(37,191) +
     ". . . HAPPY VALENTINE'S DAY . . ."+
     STRING$(6,191)

2050 REM ** SCROLL THE MESSAGE TWICE **
2060 REM ** HORIZONTALLY ACROSS SCREEN **
2070 FOR K = 1 TO 2
2080   I = 1
2090   PRINT@267, MID$(A$,1,38);
2100   T = 25: GOSUB 5000
2110   I = I + 1
2120   IF 1<= LEN(A$) THEN GOTO 2090
2130 NEXT K

2140 REM ** PUT MESSAGE IN CENTER **
2150 REM ** OF THE WINDOW **
2160 PRINT@267, MID$(A$,32);
2170 RETURN

3000 REM ** DRAW TWO HEARTS **
3010 REM ** DATA VALUES FOR SPOTS **
3020 REM ** TO BE SET **
3030 DATA 21,22,19,21,17,20,15,20,13,20
3040 DATA 11,21,9,22,9,23,9,24,9,25
3050 DATA 11,26,13,27,15,28,17,29,19,30
3060 DATA 21,31,23,30,25,29,27,28,29,27
3070 DATA 31,26,33,25,33,24,33,23,33,22
3080 DATA 31,21,29,20,27,20,25,20,23,21
3090 DATA -99,-99

3100 REM ** READ THE PAIRS OF DATA **
3110 READ X,Y
3120 REM ** CHECK FOR END OF DATA **
3130 IF X= -99 THEN GOTO 3180
3140 REM ** SET TWO POINTS **
3150 SET(X+XINC,Y): SET(X+XINC-1,Y)
3160 GOTO 3110

3170 REM ** CHECK TO SEE IF COMPLETE **
3180 IF XINC <>0 THEN RETURN

3190 REM ** ONE HEART DRAWN **
3200 REM ** ADJUST XINC AND DRAW 2ND **
3210 XINC = 78
3220 RESTORE
3230 GOTO 3110

4000 REM ** SHOOT AN ARROW **
4010 REM ** AND FLASH THE MESSAGES **
4020 REM ** THIS IS AN ARROW **
4030 B$ = "---->"

4040 REM ** SHOOT THE ARROW THREE TIMES **
4050 FOR K = 1 TO 3

4060   REM ** ARROW DISPLAY LOOP **
4070   FOR I = 0 TO 19
4080     PRINT@529,STRING$(I,32)+B$;
```

```
4Ø9Ø      REM ** TEST FOR FIRST TIME **
41ØØ      REM ** THRU K LOOP **
411Ø      IF K = I THEN GOTO 423Ø

412Ø      REM ** DISPLAY LOVE MESSAGE **
413Ø      PRINT@724," I L O V E  Y O U ";

414Ø      REM ** WHEN I IS GREATER THAN **
415Ø      REM ** TEN, FLASH THE LOVE NOTE **
416Ø      IF I>1Ø THEN T=5Ø; GOSUB 5ØØØ
417Ø      IF I>1Ø THEN PRINT@274,
          STRING$(2Ø,32);

418Ø      REM ** IF I IS GREATER THAN **
419Ø      REM ** TEN, AND THIS IS THE **
42ØØ      REM ** THIRD ARROW SHOT, FLASH **
421Ø      REM ** THE TOP MESSAGE ALSO **
422Ø      IF I>1Ø AND K=3 THEN PRINT @267,
           STRING$(4Ø,32);

423Ø      T = 5Ø: GOSUB 5ØØØ

424Ø      IF I>1Ø AND K=3 THEN  PRINT@267,
           MID$(A$,32);

425Ø    NEXT I

426Ø    REM ** BLANK OUT ARROW **
427Ø    PRINT@548,STRING$(6,32);

428Ø NEXT K
429Ø RETURN

5ØØØ REM ** DELAY ROUTINE **
5Ø1Ø FOR TT = 1 TO T
5Ø2Ø NEXT TT
5Ø3Ø RETURN
```

TRS-80 Art Program

```
1ØØ REM ** TRS-8Ø ART **
11Ø REM ** CLEAR SCREEN AND STRING SPACE **
12Ø CLS
13Ø CLEAR 1ØØ

14Ø REM ** SET ON ERROR CONDITION **
15Ø ON ERROR GOTO 1ØØØ

16Ø REM ** PAINT THE SCREEN WHITE **
17Ø FOR I = Ø TO 896 STEP 64
18Ø PRINT@I, STRING$(64,191);
19Ø NEXT I

2ØØ REM ** START NEAR THE MIDDLE **
21Ø X = 63: Y = 23
22Ø RESET(X,Y)

23Ø REM ** PICK A RANDOM DIRECTION **
24Ø REM ** TO POINT **
25Ø R = RND(8)
```

```
260 REM ** SAVE THE CURRENT LOCATION **
270 XX = X: YY = Y

280 REM ** ADJUST X,Y TO POINT AT SPOT **
290 REM ** CHECK FOR SPOT ON RIGHT **
300 IF R<4 THEN X = X + 1
310 REM ** CHECK FOR SPOT ON LEFT **
320 IF R>4 AND R<8 THEN X = X - 1
330 REM ** CHECK FOR UP **
340 IF R>2 AND R<6 THEN Y = Y -1
350 REM ** CHECK FOR DOWN **
360 IF R>6 OR R=1 THEN Y=Y + 1

370 IF INKEY$<>"" THEN 500

380 REM ** CHECK TO SEE IF SPOT IS SET **
390 IF POINT(X,Y) THEN GOTO 220

400 REM ** SPOT NOT SET, CHOOSE ANOTHER **
410 REM ** RESTORE X AND Y **
420 X = XX: Y = YY
430 GOTO 250

500 REM ** ACCEPT CHANGES TO X AND Y **
510 REM ** FIRST REQUEST X CHANGES **
520 PRINT @960, "ENTER X'S MOVE(+ OR -)";

530 REM ** USE SUBROUTINE TO ACCEPT INPUT **
540 GOSUB 2000

550 REM ** INCREMENT IS IN A **
560 XX = XX + A

570 REM ** Y CHANGE REQUEST **
580 PRINT @960, "ENTER Y'S MOVE(+ OR -)";
590 GOSUB 2000
600 YY = YY + A
610 GOTO 420

1000 RESUME 420

2000 REM ** INPUT SCAN ROUTINE **
2010 REM ** CLEAR STRING VARIABLES **
2020 A$ = "": B$ = ""

2030 REM ** BEGIN KEYBOARD SCAN **
2040 B$ = INKEY$: IF B$="" THEN 2040

2050 REM ** CHARACTER PRESSED **
2060 REM ** CHECK FOR ENTER-KEY **
2070 IF B$=CHR$(13) THEN GOTO 2170

2080 REM ** PUT CHARACTER ON SCREEN **
2090 PRINT B$;

2100 REM ** ADD CHARACTER TO STRING **
2110 A$ = A$ + B$

2120 REM ** CLEAR B$ AND GO GET **
2130 REM ** NEXT CHARACTER **
2140 B$ = ""
2150 GOTO 2040
```

```
216Ø REM ** CONVERT TO NUMBER **
217Ø A = VAL(A$)

218Ø REM ** CLEAR PRINT AREA **
219Ø PRINT @96Ø, STRING$(3Ø,32);

22ØØ RETURN
```

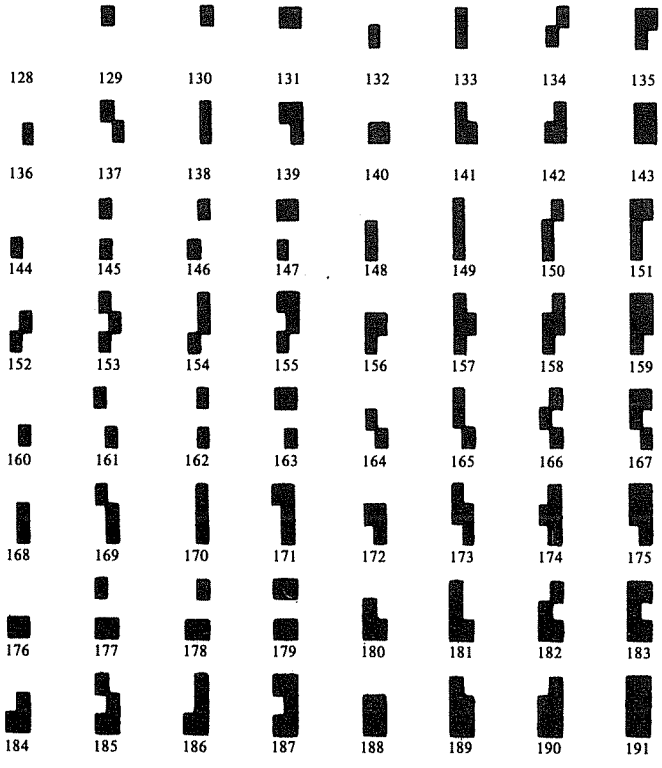
Summary

In this final chapter, you have explored combining several features of your TRS-80 (graphics, animation, and scrolling) into larger programs and you have used the POINT function, perhaps for the first time.

If you have worked your way through both books in this series, you have now covered nearly every BASIC language statement and function listed in your TRS-80 *Level II BASIC Reference Manual.* The TRS-80 computer contains many useful features that allow you to build clever and sophisticated programs. The use of the screen graphics, the error handling routines, the string functions, and the full range of mathematical operations make the TRS-80 one of the most powerful small computers on the market. We trust that these books have assisted you in using some of that power and ability. Thank you for joining us in these explorations.

APPENDIX A

Table of Graphic Characters



APPENDIX B

The Cassette Recorder

A cassette recorder is provided with the basic TRS-80 system for saving and loading programs that you write. It also is used to load taped programs that you acquire from other sources.

Level II BASIC transfers data from the recorder to the computer (and vice versa) at a rate of 500 BAUD (twice the speed of Level I). The volume setting on the recorder is very critical and should be set at a lower setting for Level II tapes than for Level I. Volume settings are discussed in more detail later in this appendix.

Saving Your Programs

If you have a program in the computer's memory that you would like to save for future use, the recorder must be connected as described in your Level II BASIC Reference Manual, "Setting up the System."

1. Put the recorder in the RECORD mode by pressing down both RECORD and PLAY buttons. Use a volume setting approximately midway (about 4 on the Radio Shack CTR-80 recorder).

The command CSAVE followed by a file (or program) name is used to record programs on tape from the computer's memory. The file name may be any alphanumeric character except quotation marks. The name is enclosed in quotes.

Examples: CSAVE "1"
CSAVE "A"
CSAVE "B"

You should always write file names on the cassette case for later reference. It is also helpful to write the index setting (of the recorder) along with the file name so that the program can be found quickly in the future.

2. On the computer, now type:

```
CSAVE "A"
```

OR whatever file name you want to use

and press the ENTER KEY.

The recorder motor should be automatically started by the computer, and the program is then recorded. After the recording is completed, the computer will turn off the recorder (hopefully—see "Recorder Problems" in this appendix). The READY message will be displayed on the screen.

```
CSAVE "A"
READY
>-
```

It is a good idea to CSAVE a program more than once on the same tape in case something should happen to one copy. This can be done by leaving the recorder in the RECORD mode and typing:

`CSAVE "A"` and pressing the ENTER key again.

Once again the computer starts the recorder motor, records the program, and turns the recorder motor off. The READY message is again displayed on the video screen. When finished, press the STOP button on your recorder.

Checking Recorded Programs

After recording a program, it is advisable to *immediately* check the recording while the original program is still in the computer's memory. Then, if there was an error in the recording, the original can be recorded again. This check can be made by following these steps:

1. Rewind the cassette to the point where the recording of your program started. REWIND and FAST-FORWARD on the CTR-80 are not under remote control. Press the REWIND button until the tape has been rewound to the desired position.
2. Press the **PLAY** button on the recorder.
3. On the computer, type:

`CLOAD? "A"` and press the **ENTER** key.

Note the ? mark

or whatever file name you are using

The computer will then compare the tape recording with the original program in the computer's memory. If there are any discrepancies, the message: BAD will be displayed on the video screen. In that case, you should CSAVE the program again. If the tapes match correctly, the recording was good, and the READY message is again displayed.

GOOD RECORDING

```
CLOAD? "A"
READY
>-
```

BAD RECORDING

```
CLOAD? "A"
BAD
READY
>-
```

Loading Taped Programs

The volume setting recommended by Radio Shack for loading cassette tapes on a Level II system with the CTR-80 tape recorder is approximately 4.

For prerecorded tapes by other manufacturers, you will probably have to experiment to find the volume setting at which they will load correctly. (See "Recorder Problems" in this appendix for help.)

To transfer a prerecorded program to the computer's memory:

1. Connect the recorder as discussed in your Level II Reference Manual.
2. Insert the cassette containing the prerecorded program and rewind the cassette to the beginning of the desired program.
3. Press the PLAY button on the recorder.
4. On the computer, type:

```
CLOAD "A"
```

or the file name of the program that you want

and press the **ENTER** key.

The computer turns on the recorder's motor. After a few seconds, two asterisks should appear in the upper right corner of the screen. The right asterisk should blink (usually at an irregular rate). When the program has been completely transferred, the computer stops the recorder's motor and gives its READY message on the screen.

```
CLOAD "A"
READY
>-
```

If all went well, turn off the recorder. The program is loaded. Type:

```
RUN
```

Happy computering!!

Recorder Problems

The audio cassette recorder provides an inexpensive means to store programs and data files outside the computer. Keep in mind that the recorder was not originally designed for this purpose. It is a cheap alternative to floppy disk drives. Although far from perfect, it does beat loading programs from the keyboard each time that you want to use them.

Use of the tape recorder for digital purposes requires great patience, understanding, and care. Here are some problems that you may encounter, and some suggestions that may prove helpful. You may not agree with all of them.

— Use high-quality, certified digital tapes. Poor-quality audio tapes may have imperfect magnetic coatings with some spots that would be undetectable when playing music or voices. However, a loss of one bit of data during the save or load of digital data may spoil a good program. Radio Shack recommends its special 10 minute per side computer tape cassettes.
— Keep the heads of your recorder clean and demagnetized. Special tapes are available that can quickly be "run through" the recorder to clean or demagnetize the heads. Special cleaning liquids are also available.
— Be consistent in using a given volume setting when recording and loading your own taped programs.
— Use short tapes and record only one program on each side of the tape.
— Record a given program several times on the tape.

If you follow the above suggestions, only minor problems should arise when loading or saving your own tapes.

Big problems may arise, however, when you try to load tapes from other sources. You must patiently try different volume settings until a good load is accomplished. Recording levels will vary from manufacturer to manufacturer and from tape to tape. Levels have even been known to vary within a given taped program.

— Once you correctly load a tape from an outside source, make a copy of it by saving it with your own volume settings.
— Don't hesitate to return a program that you have purchased if you cannot get it to load correctly. Not all preprogrammed tapes on the market are perfect, as mass duplication of tapes is as yet an "imperfect art." The return rate is high.

There are products on the market that enhance the recorded data. They make it possible to load tapes within a wide range of volume settings. Some are done by hardware (a "black box" attached between the computer and recorder) and others are done by software (which must be loaded in from a cassette).

Inside the TRS-80 lives a relay that is supposed to turn the recorder's motor on or off at the correct time. But relays have been known to stick, so that sometimes the recorder may keep on turning after a program has been successfully loaded. At other times, you may punch the PLAY button on the recorder and find that the recorder reaches the beginning of a prerecorded program before you can type in CLOAD.

— If this condition persists, don't hesitate to have your Radio Shack store replace the relay (within the warranty period, if possible).

APPENDIX C

ERROR Codes and Messages

Message	*Error*	*Number*	*Explanation*
BS	Subscript out of Range	9	An attempt was made to assign a matrix element with a subscript beyond the DIMensioned range.
CN	Can't Continue	17	A CONT was issued at a point where no continuable program exists (as after a program was ENDed or EDITed).
DD	Redimensioned Array	10	An attempt was made to DIMension a matrix that had been previously dimensioned. It is good practice to put all DIM statements at the beginning of your programs.
FC	Illegal Function Call	5	An attempt was made to execute an operation using an illegal parameter.
FD	Bad File Data	22	Data input from an external source (such as tape) was not correct or was in improper sequence, etc.
ID	Illegal Direct	12	The use of INPUT as a direct command.
LS	String Too Long	15	A string variable was assigned a string value that exceeded 255 characters in length.
L3	Disk BASIC only	23	An attempt was made to use a statement, function, or command that is available only when the TRS-80 Mini Disk is connected via the Expansion Interface.

MO	Missing Operand	21	An operation was attempted without providing one of the required operands.
NF	NEXT without FOR	1	NEXT is used without a matching FOR statement. Also occurs if NEXT variable statements are reversed in a nested loop.
NR	No RESUME	18	End of program reached in error-trapping mode.

Message	*Error*	*Number*	*Explanation*
OD	Out of Data	4	A READ or INPUT # statement was executed with insufficient data available. DATA statement may have been left out or all data may have been read from tape or DATA.
OM	Out of Memory	7	All available memory has been used or reserved. Can be caused by large matrix dimensions or nested branches.
OS	Out of String Space	14	The amount of string space allocated was exceeded.
OV	Overflow	6	A value that was input or was derived is too large or too small for the computer to handle.
RG	RETURN without GOSUB	3	A RETURN statement was encountered before a matching GOSUB was executed.
RW	RESUME without error	19	A RESUME was encountered before ON ERROR GOTO was executed.
SN	Syntax Error	2	This usually is the result of incorrect punctuation, an open parenthesis, an illegal character, or a misspelled command.
ST	String Formula Too Complex	16	A string operation was too complex to handle. Break up the operation into shorter steps.
TM	Type Mismatch	13	An attempt was made to assign a nonstring variable to a string or vice versa.

UE	Unprintable Error	20	An attempt was made to generate an error using an ERROR statement with an invalid code.
UL	Undefined Line	8	An attempt was made to refer or branch to a nonexistent line.
/0	Division by Zero	11	An attempt was made to divide by zero.

NOTE: Some errors are difficult to locate. A program line may look correct on the video screen but contain a "hidden error." For example: A SHIFTed character may have been typed where an unSHIFTed character was required. (Such as a SHIFTed @ in PRINT @ or a SHIFTed variable). Spaces are sometimes important. Examine the line carefully for places where spaces should be inserted. If you can't find anything wrong with the line which causes an error, try retyping the line. Take care to avoid the SHIFT key unless it is required.

Program Index

Index

TRS-80™

Advanced Level II BASIC

a learning guide

■ Sample programs ■ Chapter summaries ■ Test questions ■ Animation ■ Graphics
■ Tips on saving memory space ■ Arithmetic functions ■ Error handling routines
■ PEEK and POKE commands ■ Review of BASIC terms, statements and commands
■ Scrolling ■ Sound ■ File handling techniques ■ Numeric and string arrays

Learn to use advanced programming techniques on your TRS-80 computer!

Now you can go beyond beginner's BASIC! Co-authors Don Inman, Bob Albrecht, and Ramon Zamora once again combine their computer programming talents as they did in their *TRS-80 Level II BASIC* self-teaching guide. But in this book, they take you step-by-step through a number of the *advanced* features of Level II BASIC. It's easy to read and understand so you'll quickly master the higher programming techniques, allowing you to take full advantage of the many capabilities of the TRS-80 Level II computer.

Printed in U.S.A. for Radio Shack A Division of Tandy Corporation

ISBN 0 471 09677-6